Reflection with First-Class Implementations

Turtling down Runtime Meta-levels

and PCLSRing up

François-René Rideau, *TUNES Project*

Lisp NYC, 2017-03-21

http://fare.tunes.org/files/cs/fci-ln2017.pdf

Presented at BostonHaskell (2016), rejected at SNAPL 2017

This Talk

A vision, with sound theory, but unimplemented.

A research program for me — and I hope for you.

Salvaged from my aborted 1999 PhD thesis:

The Semantics of Reflective Systems

and beyond that, my TUNES project

At ENS, P. Cousot taught Abstract Interpretation

All I was interested in was the opposite direction:

Concrete Implementation

The Basic Intuition

"Good programmers can zoom in and out

of levels of abstraction, understanding that

none possibly contradicts the other ones."

What if you could navigate those levels
at runtime?

... so you don’t have to be a genius who can do it all in your head before compile-time...

... what else could you do with this super-power?

The Take Home Points

Formalizing Implementations: Categories!

Observability: Neglected key concept — safe points

First-Class Implementations via Protocol Extraction

Explore the Semantic Tower — at runtime!

Principled Reflection: Migration

Natural Transformations generalize Instrumentation

Reflective Architecture: 3D Towers

Social Implications: Platforms, not Applications

Plan

Formalizing Implementations

First-Class Implementations

Principled Reflection

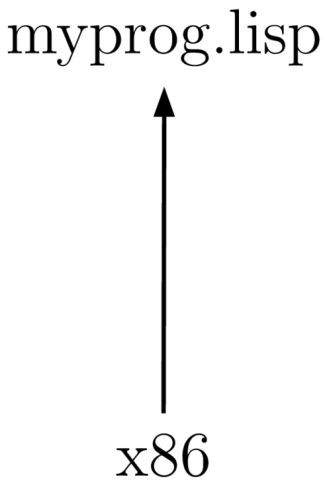
Reflective Architecture

I. Formalizing Implementations

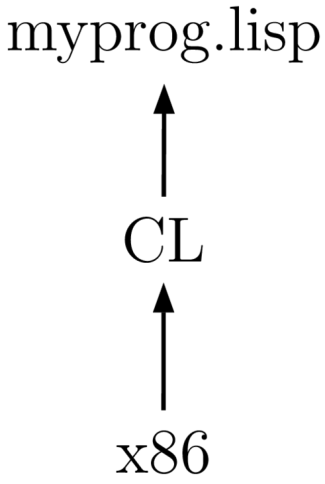
I.1 A Universal Framework

Implementations, informally

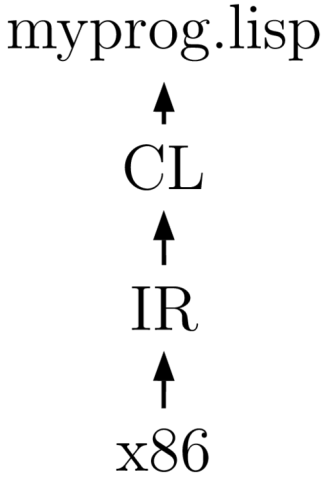
To implement a Lisp program on a PC...



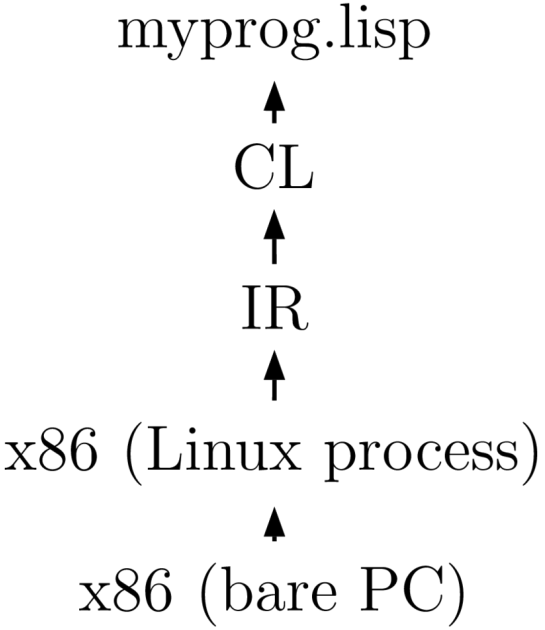
First, define (Common) Lisp...



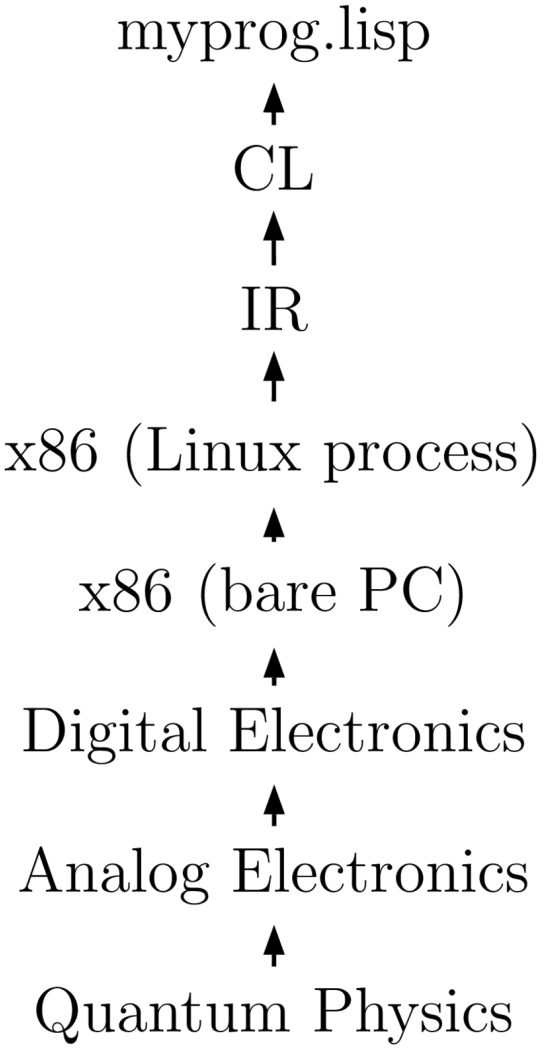
Lisp is hard, reduce it to some IR...



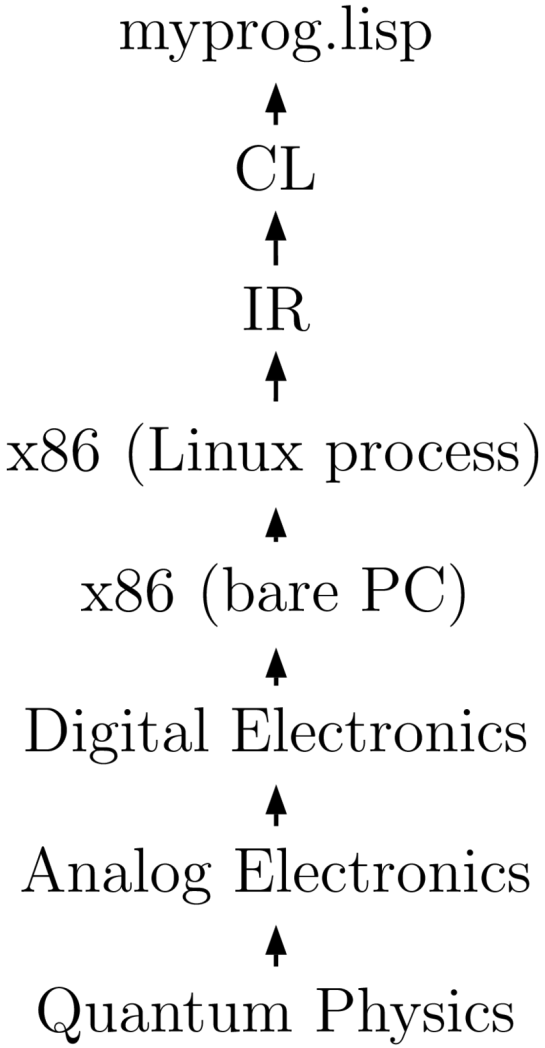
What do you mean, x86?



There is no bottom!



Always finer divisions



Implementations, informally

To implement a program on a computer...

Goal: to relate two computations

... via SBCL 1.3.15, using Linux

Hold together *Towers* of computations

Can we *reason* about implementations?

Basic correctness, other useful properties

Formalization Challenges

First, must formalize computations

Few are adequately formalized

When they are, with incompatible formalisms

How can we unify these formalisms?

What suitable relation between two computations?

What composable properties for these relations?

Unify Existing Semantic Models

Operational Semantics (Small Step)

Operational Semantics (Big Step)

Labeled Transition Systems

Term Rewriting, Rewrite Logic

Modal Logic

Partial Order

Abstract State Machines

...

even Denotational Semantics (2 ways)

Category Theory

A category: nodes, arrows, structure

Mathheads: node=object arrow=(homo)morphism

Structure: equality, identity, composition — *ad lib*.

Functor: relate two categories, preserving structure

Higher: (categories + functors) as (nodes + arrows)

Natural Transformations: functors between functors

Why Category Theory?

Simple core

Unlimited abstraction

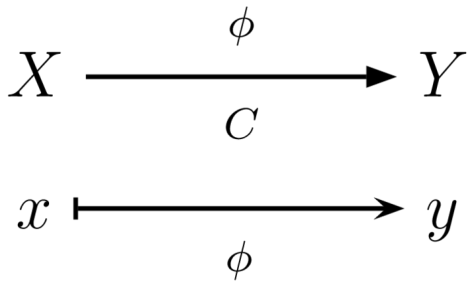
Universal: graphs, preorders, labeled transitions...

Structure preservation: theorems "for free"

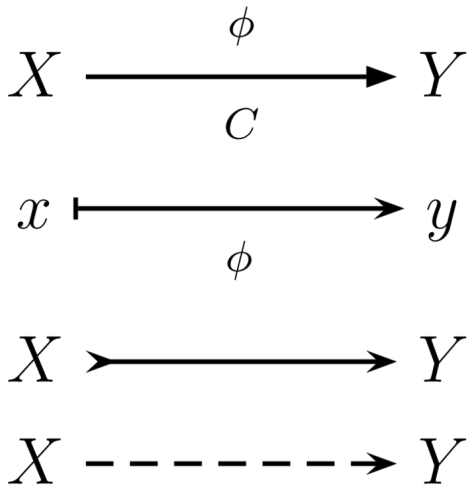
Types, Curry-Howard Isomorphism

Better foundation than Set Theory

Categories



Categories



Computation as Categories

Nodes: states of the computation

Arrows: transitions between states, traces

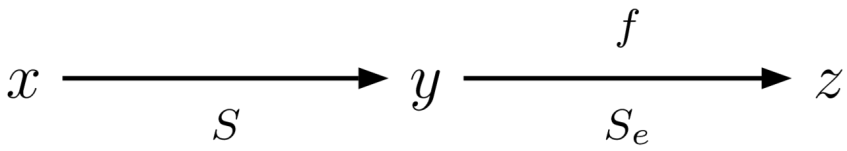


Figure conventions:

Computation time goes left to right

Label above, (sub)category 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)

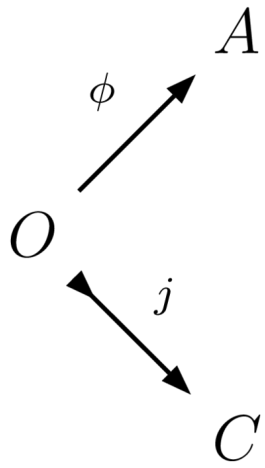
Co-functorial vs Functorial

Noisy vs lossy

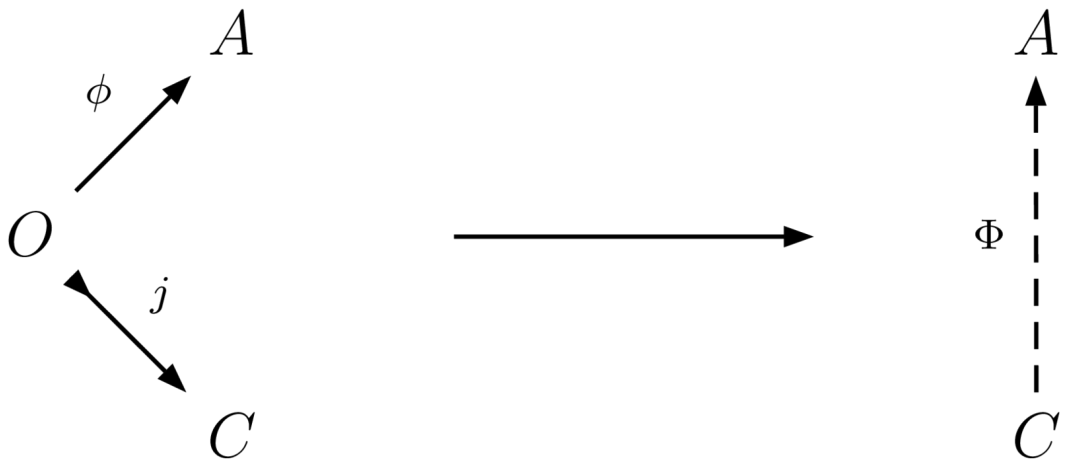
Non-deterministic vs deterministic

Partial Functions (1)

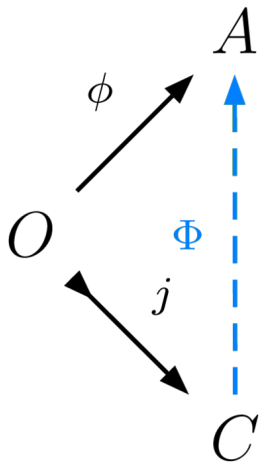
Partial Functions (2)



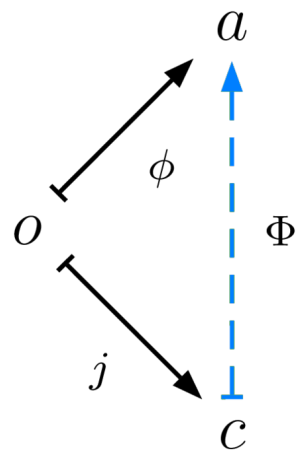
Partial Functions (3)



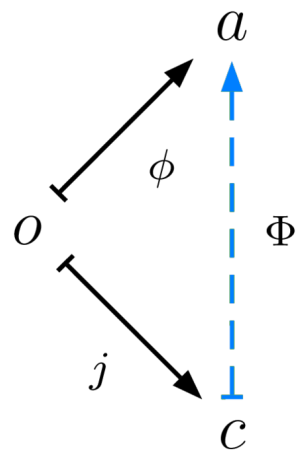
Deduction



Observable State



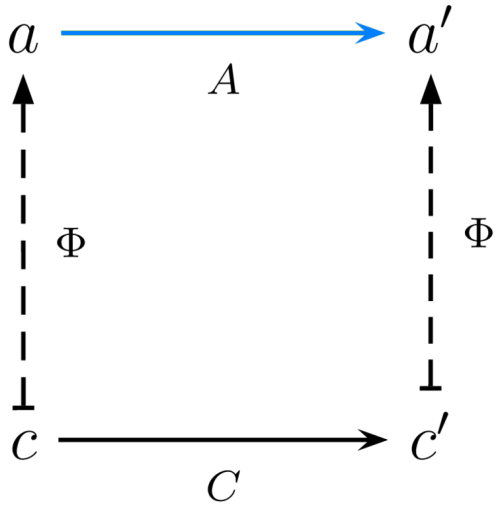
Observable State



$o = c$

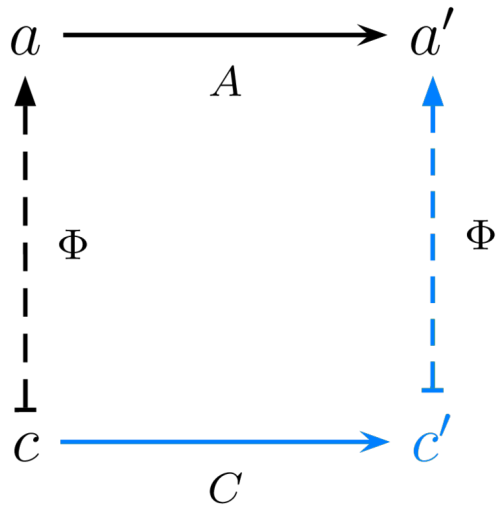
I.2 Properties of Implementations

Soundness

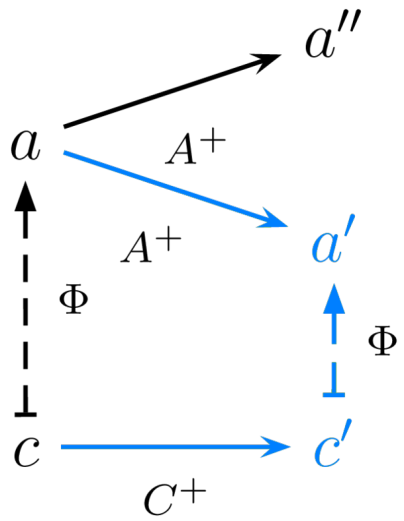


Totality

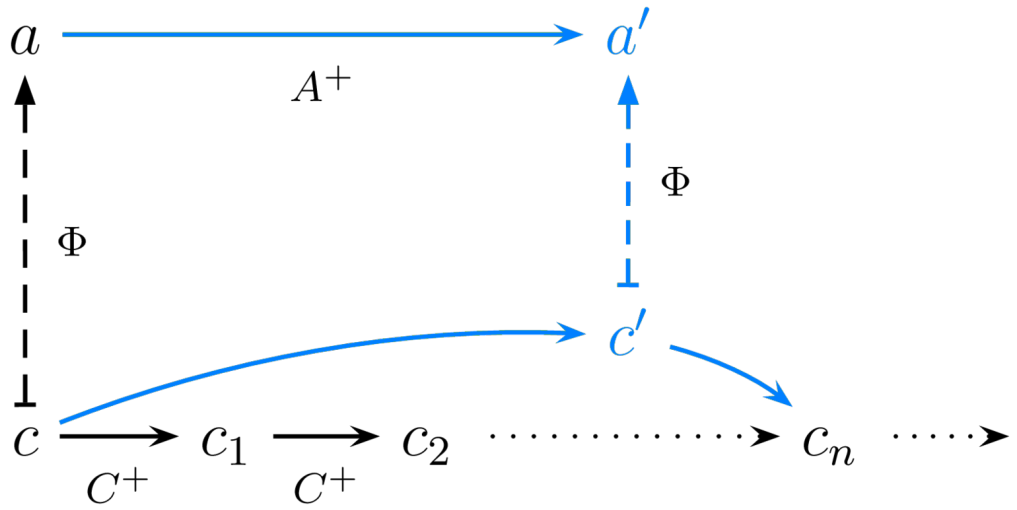
Completeness



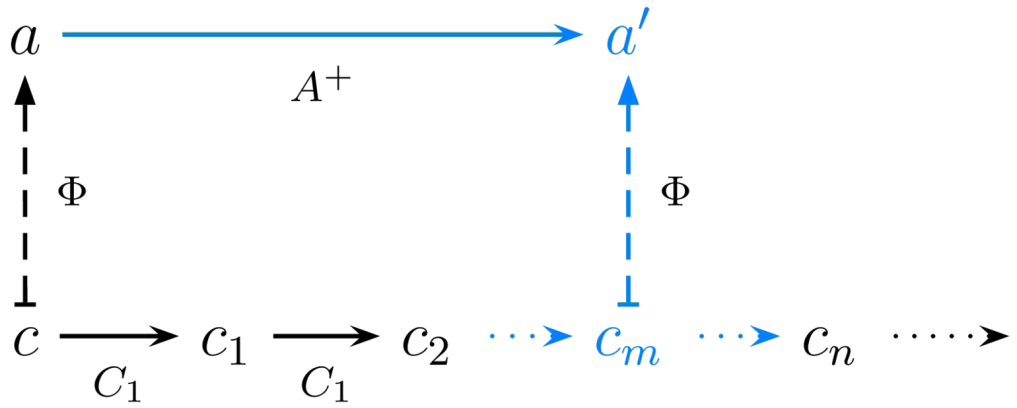
Advance Preservation



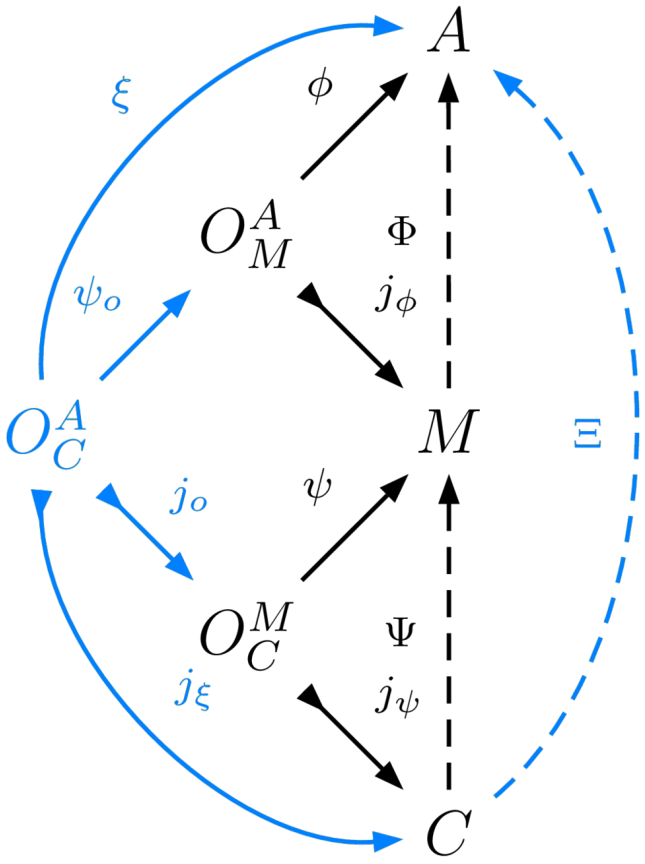
Liveness



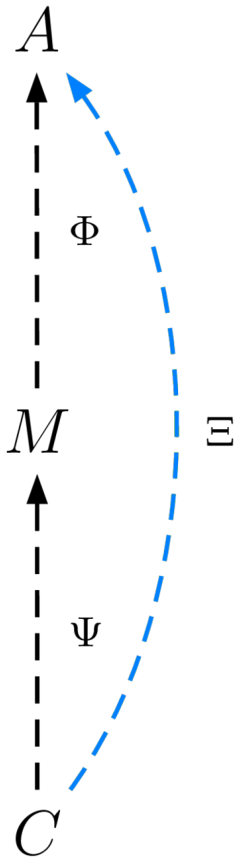
Strong Liveness



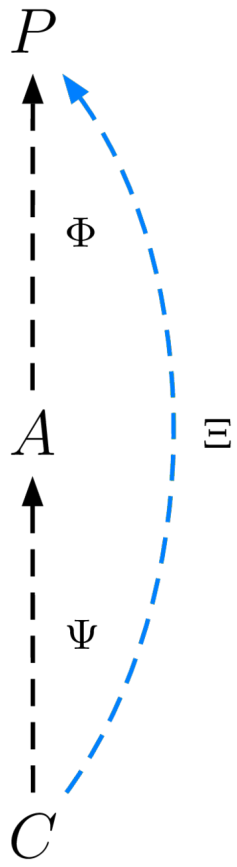
Composability



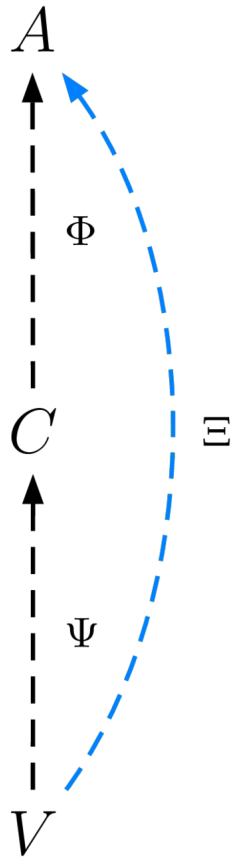
Composability



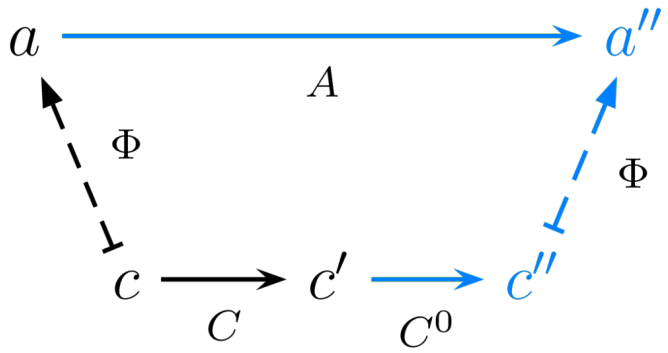
Composability



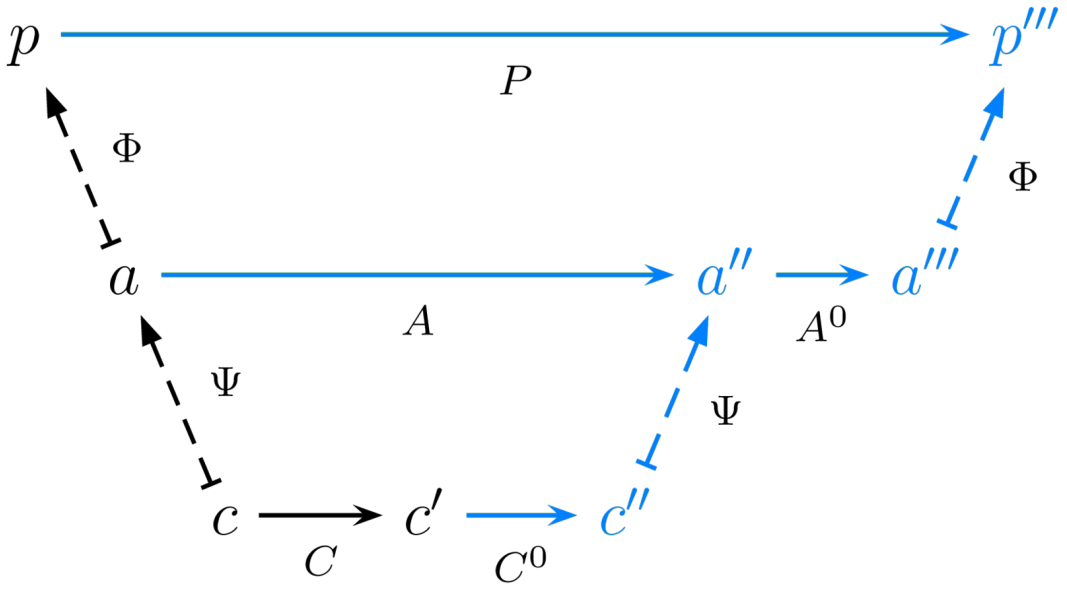
Composability



Observability (aka PCLSRing)

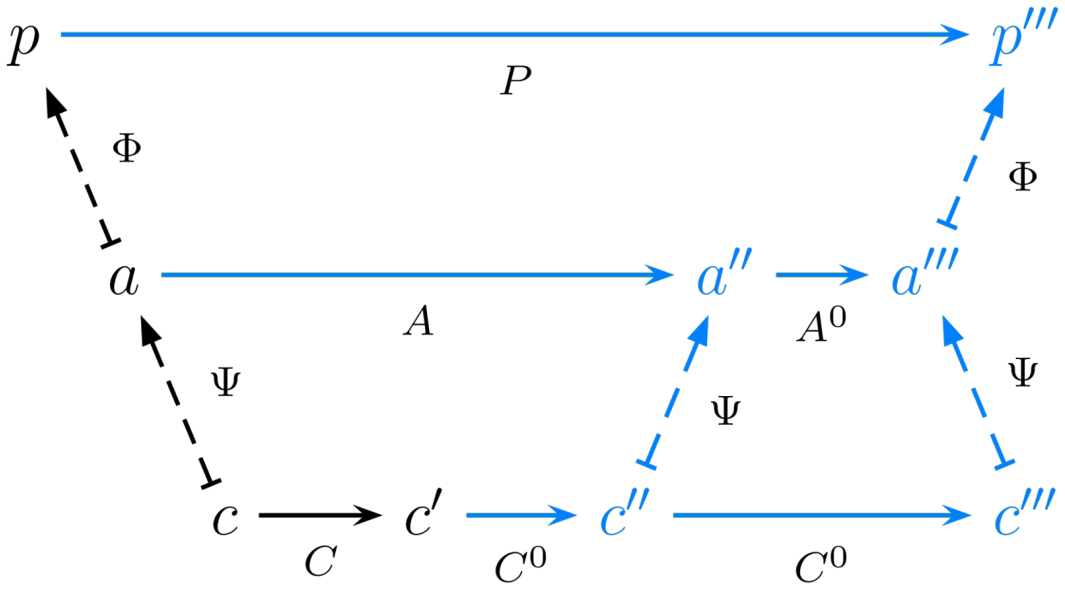


Observability (aka PCLSRing)



... not composable!

Observability + Completeness



Composable!

II. First-class Implementations

II.1 Protocol Extraction

Protocol: Categories (in Agda)

```
record Category … : Set … where …
  field
    Obj : Set o
    _⇒_ : Rel Obj a
    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)
```

Protocol: Operational Semantics

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

Usual functions: →

Effectful functions: ⇢ (non-det)

Protocol: Operational Semantics

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

  eval :: s ⇢ Arr s
  advance :: s ⇢ Arr s
```

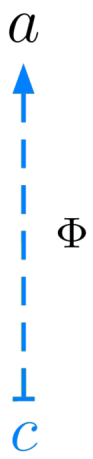
Protocol: Implementation

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

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

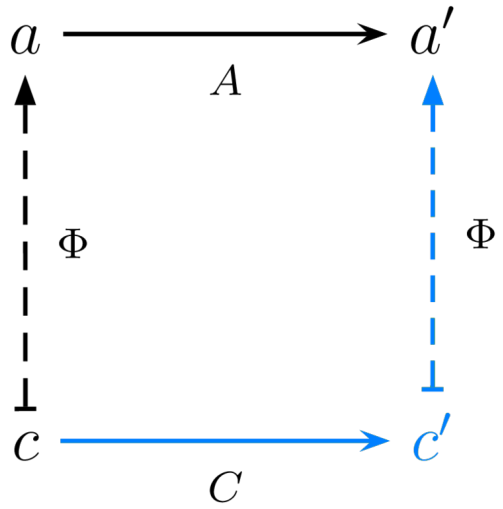
Arr = pirate sound = functorial map

Protocol: Totality



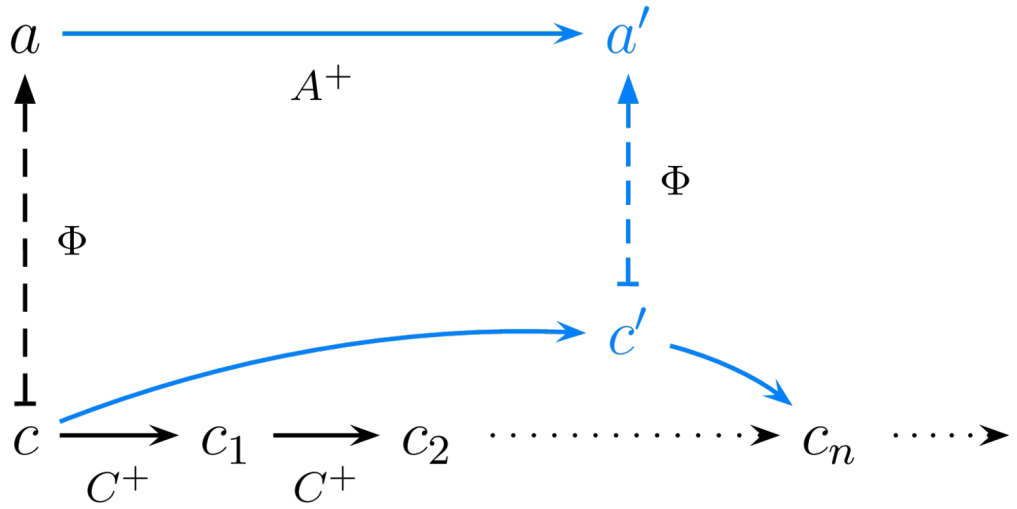

```
implement :: a ⇢ c
```

Protocol: Completeness



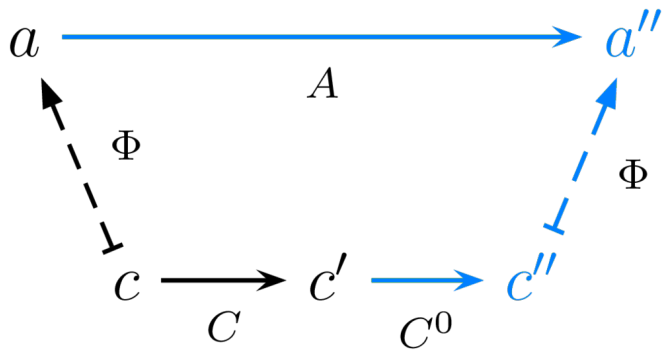

```
implementArr :: c → (Arr a) ⇢ (Arr c)
```

Protocol: Liveness




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

Protocol: Observability (PCLSRing)




```
safePoint :: c ⇢ Arr c

pedanticSafePoint :: 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:

```
record :: m ⇢ s
recordArr :: 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 an =
  c.perform (implement an)
a.performArr aa state =
  c.performArr (implementArr (c.record state) aa) state

a.record state =
  interpret (c.record m)
a.recordArr state change =
  interpretArr (safePoint (c.record state change))
```

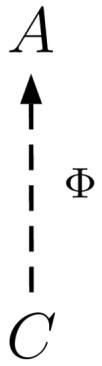
Lifting Evaluation Protocols

If the implementation is live, observable:

```
a.run an =
  interpretArr (safePoint (c.run (implement an)))
a.advance an =
  interpretArr (advanceInterpretation (implement an))
```

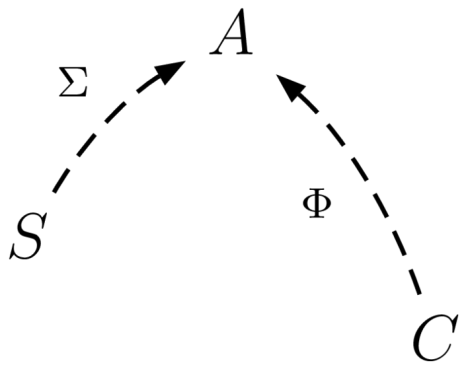
II.2 The Semantic Tower

Compilation (1)



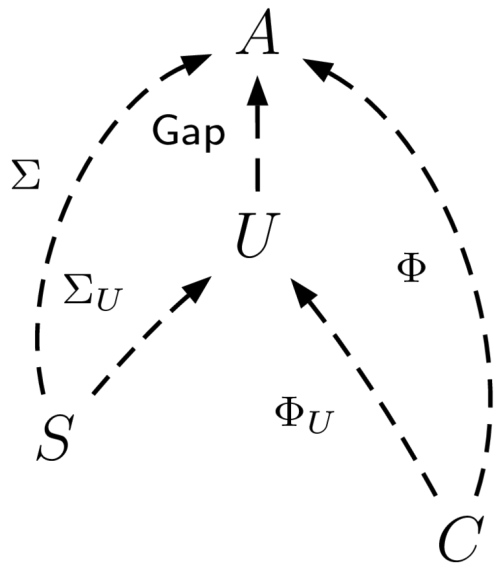

```
implement :: (Impl a c) ⇒ a ⇢ c
```

Compilation (2)



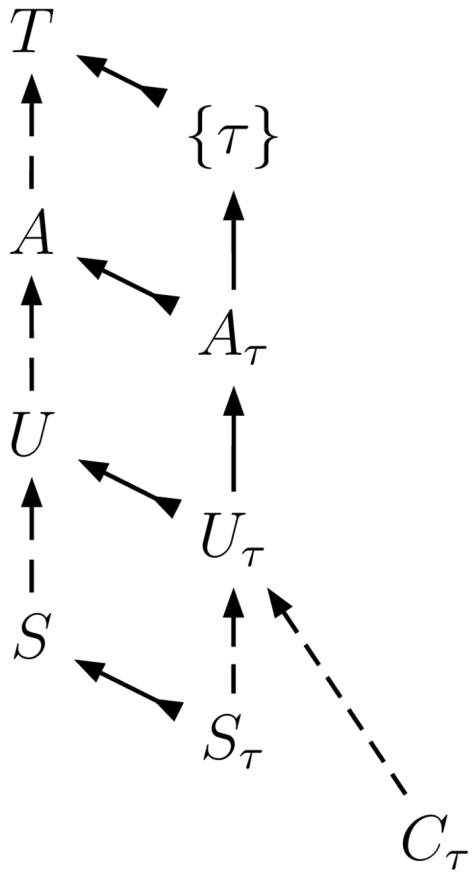

```
interpret :: (Impl a s) ⇒ s ⇢ a
implement :: (Impl a c) ⇒ a ⇢ c
```

Compilation (3)



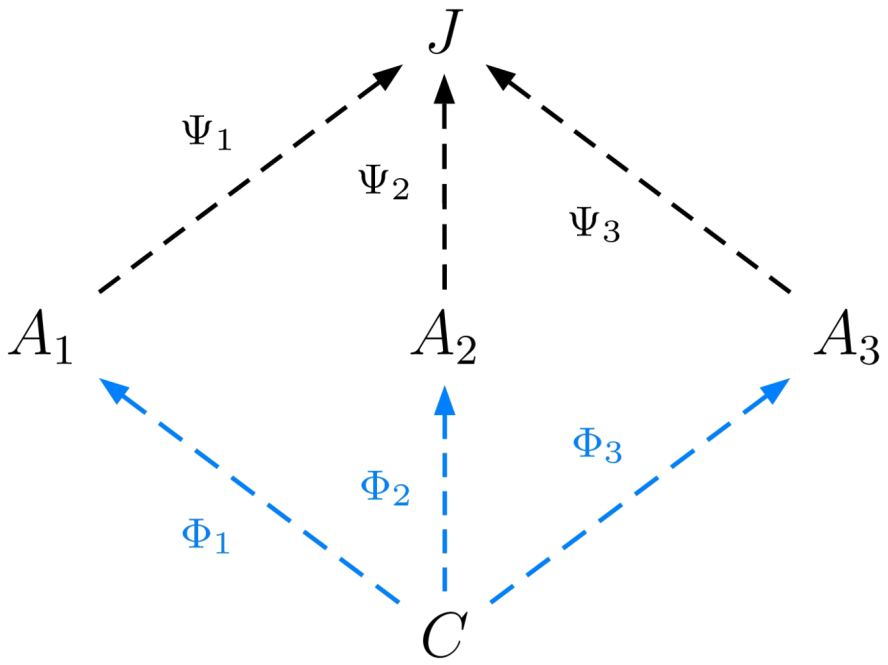

```
u :: OpSem -- specify up to what rewrites
interpret :: (Impl u s) ⇒ s ⇢ u
implement :: (Impl u c) ⇒ u ⇢ c
```

Static Type Systems

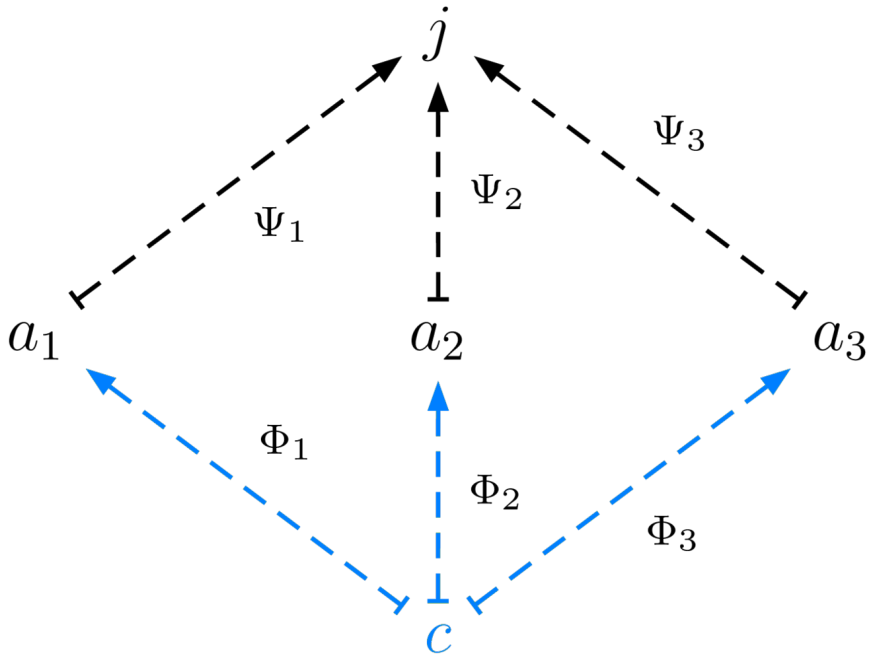


Subject reduction: *T* contains no exomorphisms

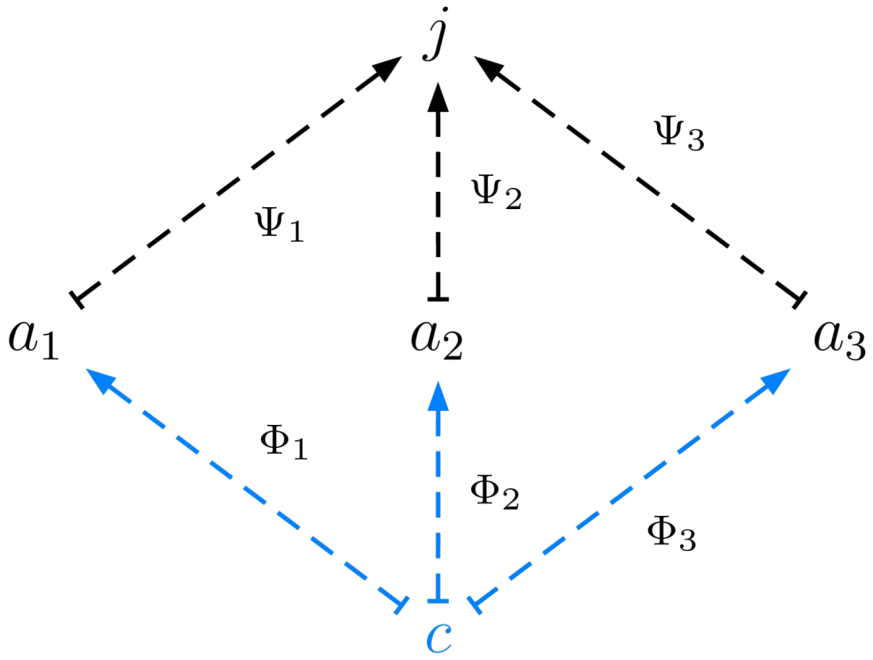
Aspect-Oriented Programming (1)



Aspect-Oriented Programming (2)

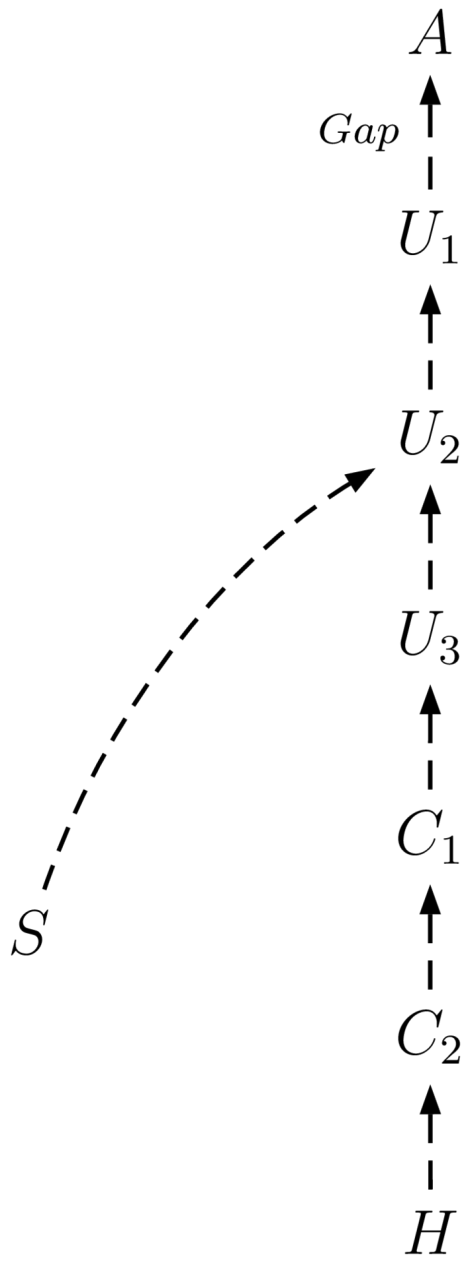


Aspect-Oriented Programming (2)

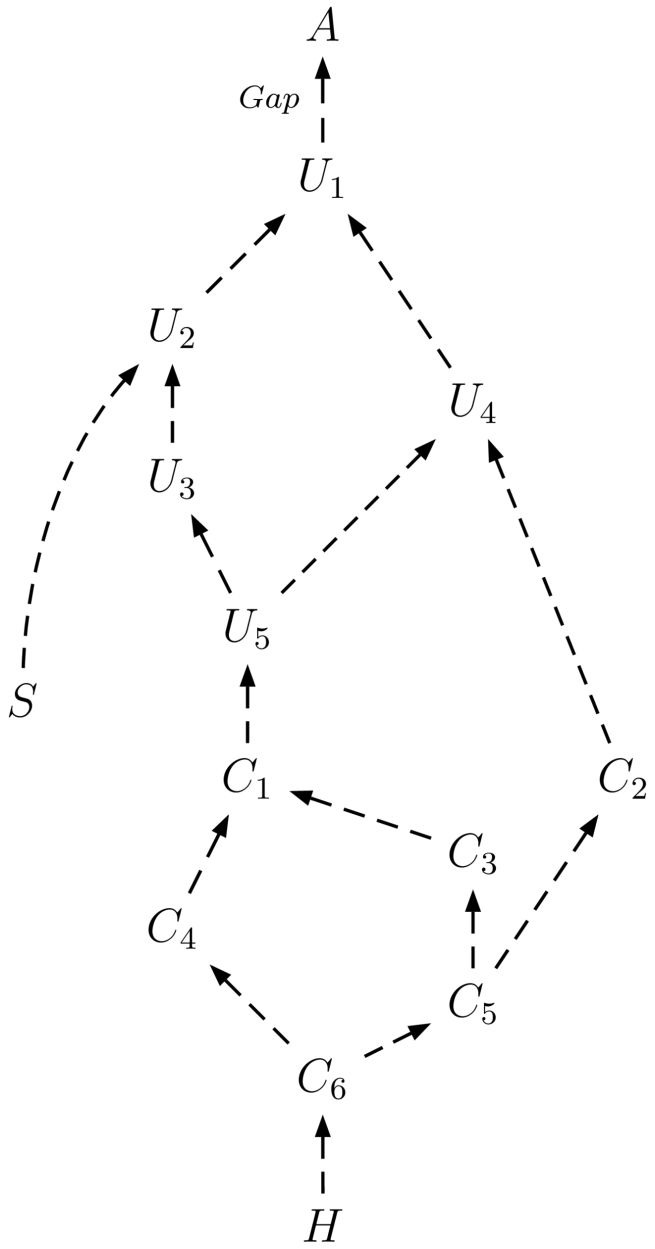


Constraint Logic Meta-programming!

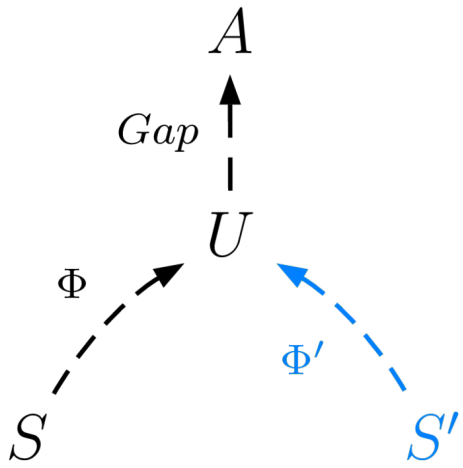
Semantic Tower



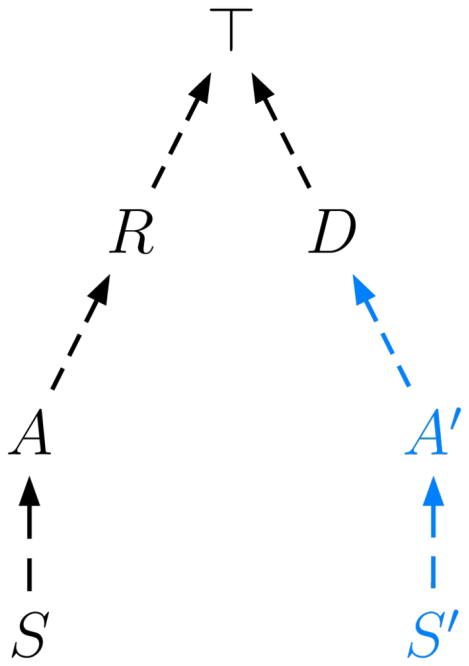
The Tower is not Linear



Refactoring



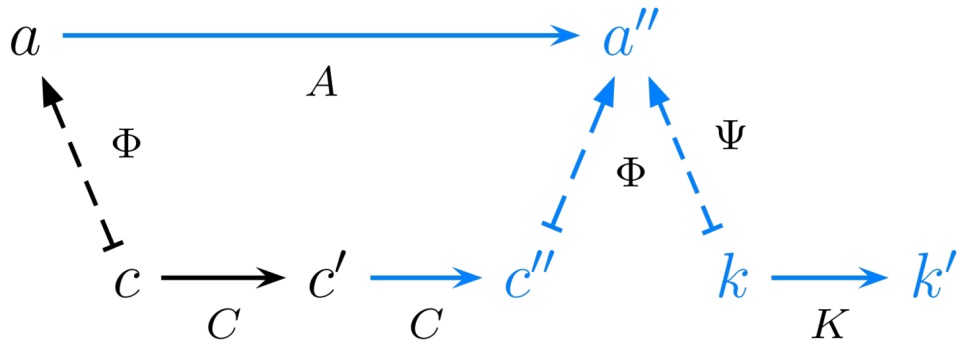
Developing



III. Principled Reflection

III.1 Migration

Migration



When your hammer is Migration...

Process Migration

Garbage Collection

Zero Copy Routing

Dynamic Configuration

JIT Compilation

etc.

Requirement: Full Abstraction

Computations have a clear opaque bottom:

1- It’s perfectly clear what the bottom is

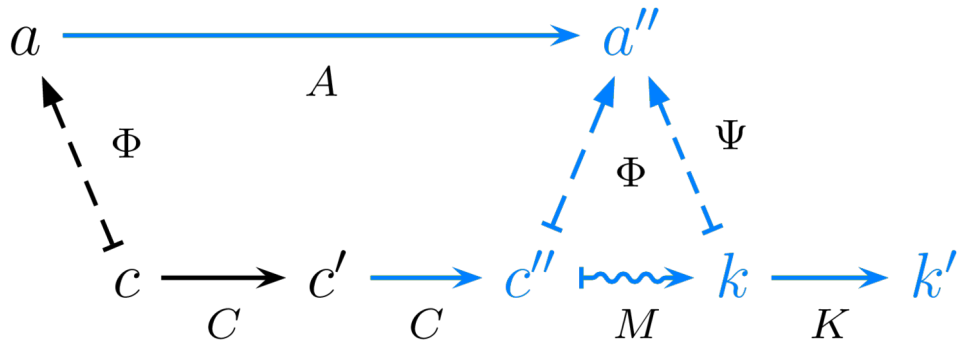
2- The bottom is totally opaque

Indeed, what’s below can change at runtime!

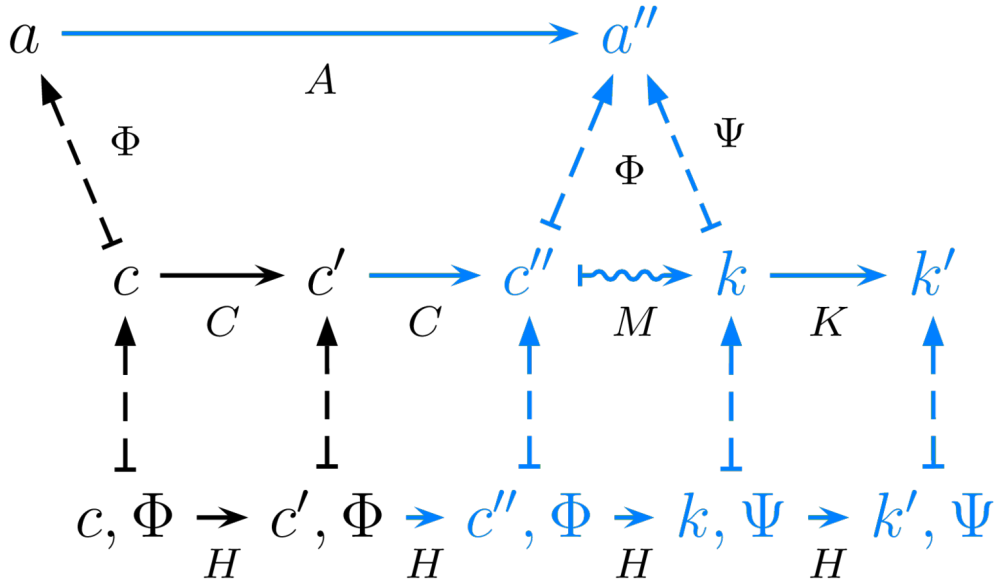
Alternatively, include what’s "below"

The language or system must explicitly support that

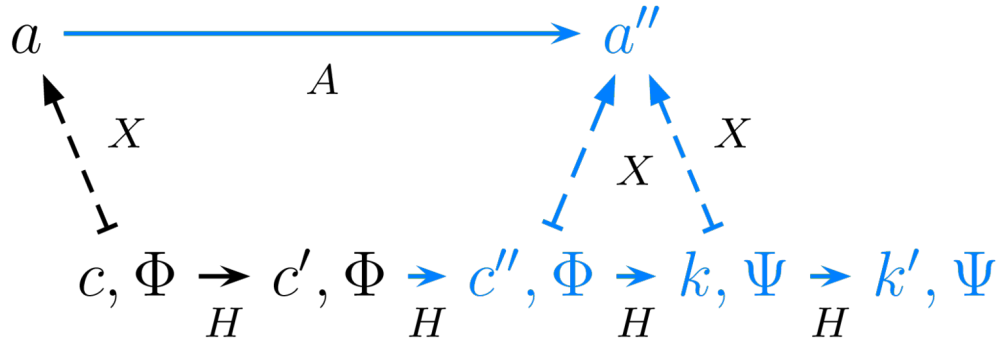
Migration (Optimized)



Migration (Implemented)



Migration (Factored out)



Fruitful change in Perspective

Correctness

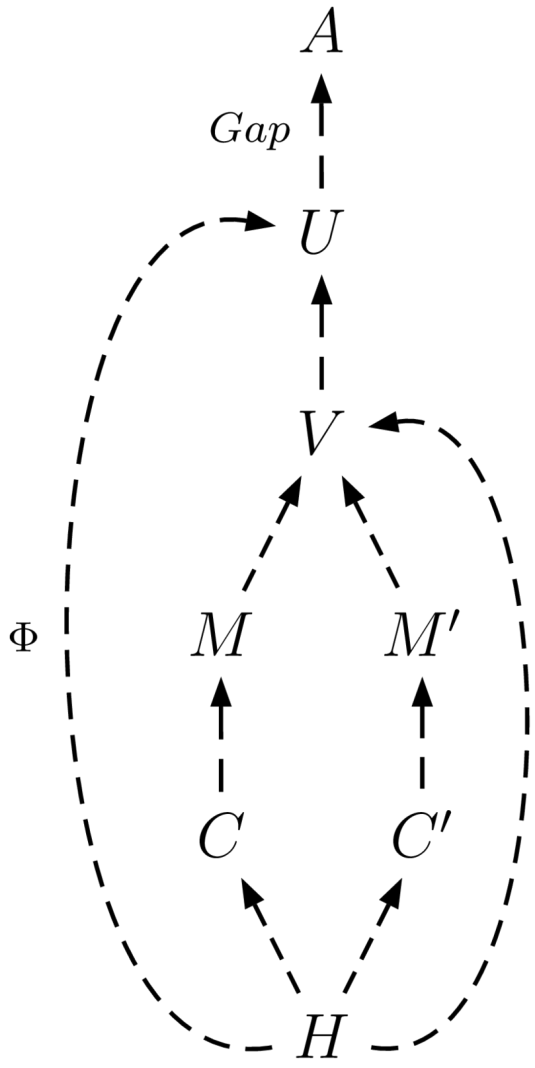
Dynamism

Retroactivity

Composability

Predictable Cost-Reduction

Migration Tower



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

Another dimension to diagrams! Turtles?

Implicit I/O

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

Handled by 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

More like Emacs libraries and browser plugins

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 (redux)

Formalizing Implementations: Categories!

Observability: Neglected key concept — safe points

First-Class Implementations via Protocol Extraction

Explore the Semantic Tower — at runtime!

Principled Reflection: Migration

Natural Transformations generalize Instrumentation

Reflective Architecture: 3D Towers

Social Implications: Platforms, not Applications

Challenge

Put First-class Implementations in your platform

Platform: PL, IDE, OS, Shell, Distributed System

Factor your software into meta-levels

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

Thank you!

My blog: *Houyhnhnm Computing*

http://ngnghm.github.io/