
# ContextAmber

## A COP implementation for Amber Smalltalk
## Seminar Context-oriented Programming, WS2014/15

Matthias Springer

Hasso Plattner Institute, Software Architecture Group

January 13, 2015

# Overview

# Introduction

- Context-oriented Programming: modularize heterogeneous crosscutting concerns
- Layers, partial methods, dynamic layer activation at runtime
- ContextAmber: a COP implementation for Amber Smalltalk (Smalltalk runtime environment in the web browser)
- Challenge: make it fast by inlining partial methods (with object-wise layer activation!)
- Use case: Debug output for Athens vector graphics library

# Related Projects

- ContextJS [2]: COP implementation for JavaScript, written in JavaScript
- SqueakJS [1]: Smalltalk environment running in the web browser
- Athens: Vector graphics library for Pharo developed by Igor Stasenko
  - Original Pharo implementation:
    `http://smalltalkhub.com/#!/~Pharo/Athens`
  - Implementation for Amber Smalltalk using HTML5 canvas:
    `https://github.com/matthias-springer/amber-athens`

# Overview

# Drawing Simple Paths with Athens

# Drawing Simple Paths with Athens

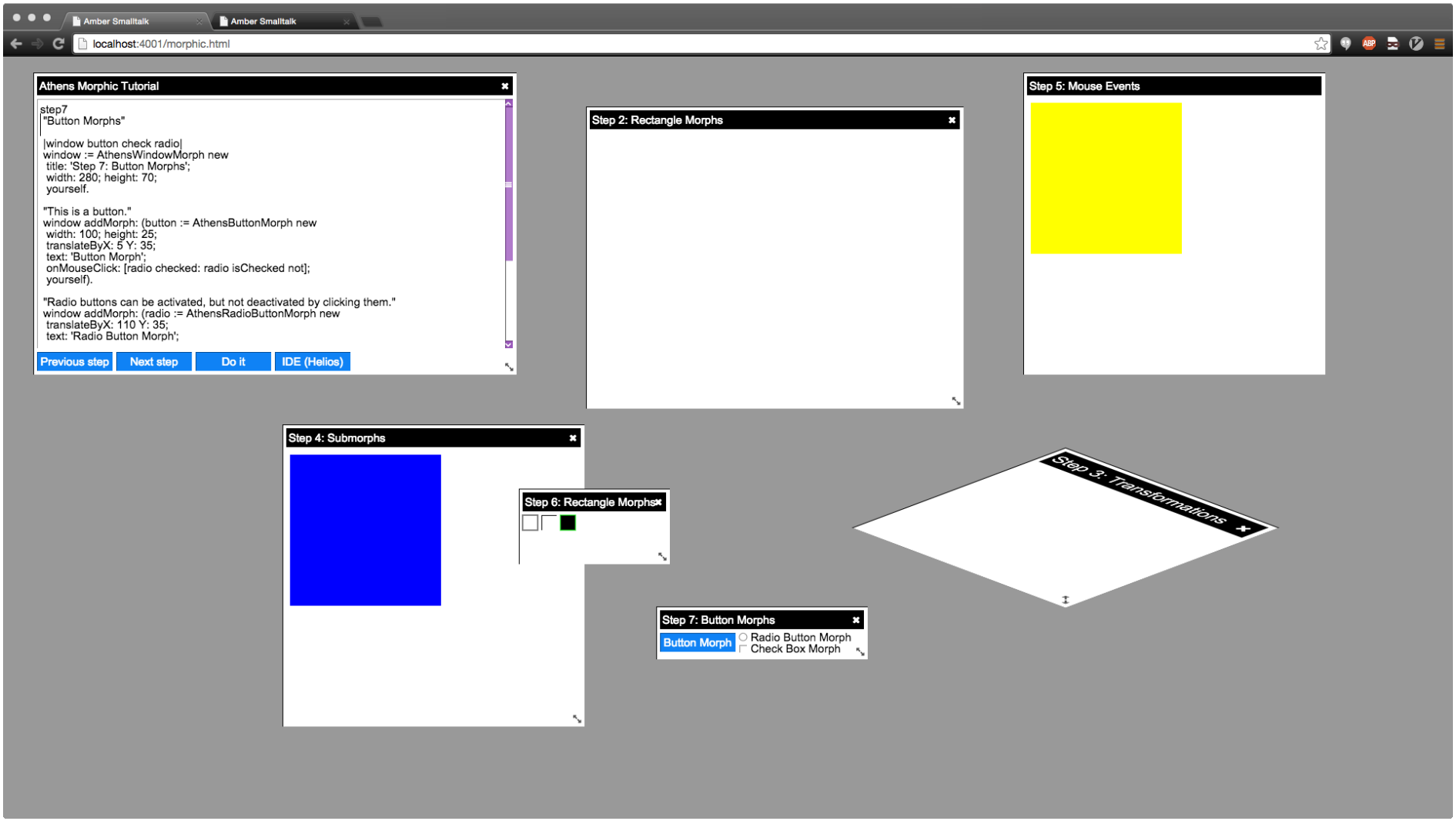
```
step14
    | path |
    path := surface createPath: [ :builder |
        builder
            absolute;
            lineTo: -50@ -50;
            "quadric Bezier curve"
            curveVia: 0@ -80 to: 50@ -50;
            "cubic Bezier curve"
            curveVia: 100@ -20 and: -50@20 to: 50@50;
            "clockwise arc"
            cwArcTo: 50@100 angle: 45;
            "counter-clockwise arc"
            ccwArcTo: -50@100 angle: 45.
        builder close ].

    surface drawDuring: [ :canvas |
        surface clear: Color gray.
        canvas setShape: path.
        canvas draw ].
```

# Drawing Simple Paths with Athens

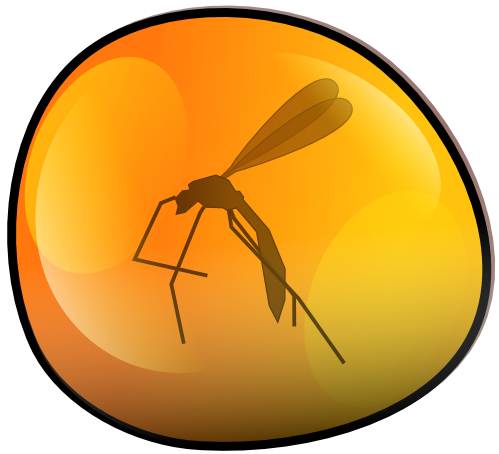
- `ControlPointLayer`: show control points for Bézier/. . . curves, start point and end point for all path segments, all path movements
- `TangentLayer`: show tangents/derivatives for curves at control points
- Activate layers on a per-path basis, but also globally
- Performance criteria: frames per seconds

# Drawing Morphs with Athens



# Overview

# Amber Smalltalk



- Client-side web application framework for dynamic JavaScript-based web applications
- Smalltalk to JavaScript compiler and small/tidy Smalltalk standard library
- No image: classes/runtime environment are initialized during startup
- Built-in IDE: legacy IDE and Helios IDE

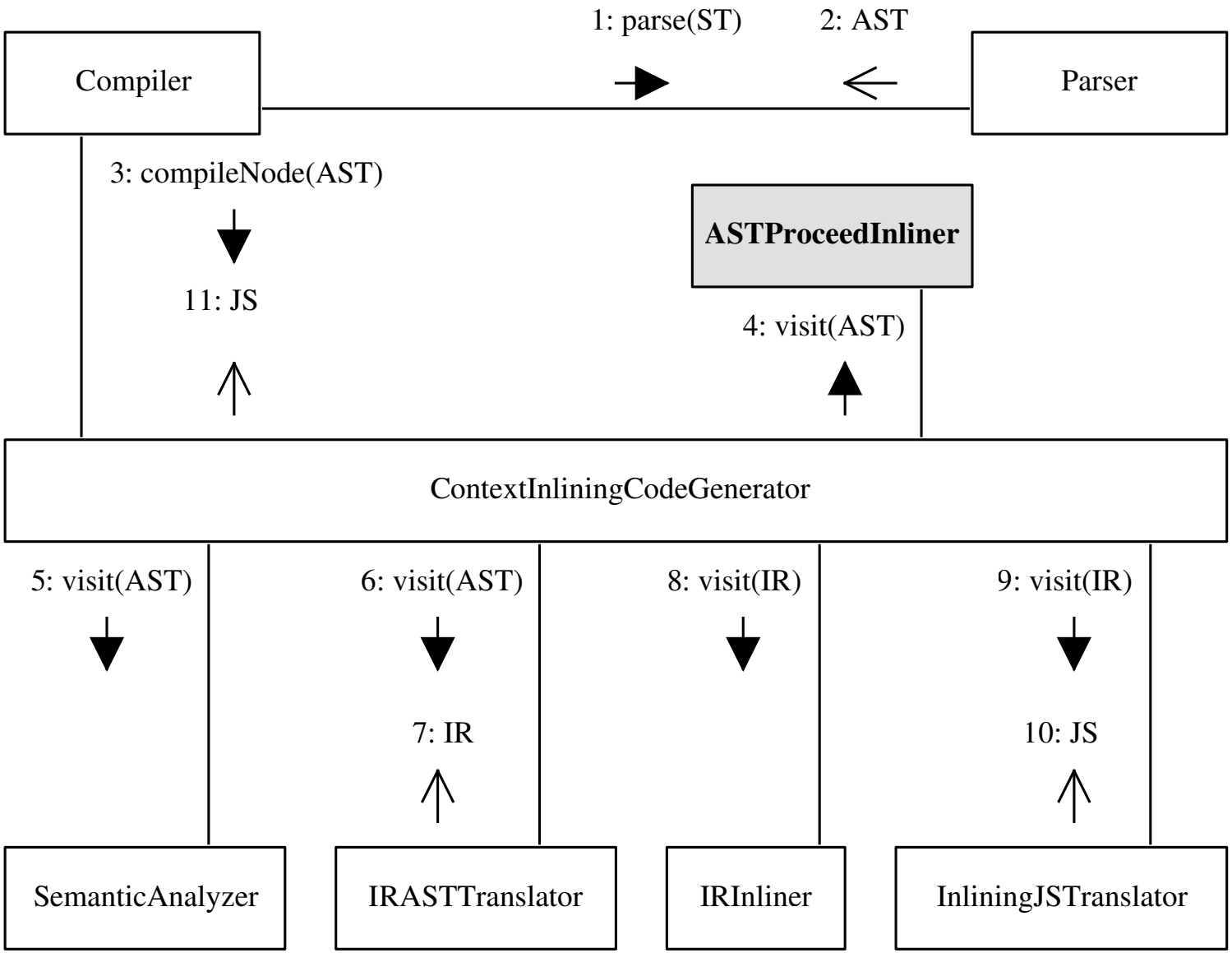
# Why Amber Smalltalk?

- Nice framework, used by real people, good community
- Few classes, small project, easy to understand
- Well-structured object model and compiler
- **Easy to prototype new ideas**

# Smalltalk to JavaScript Compilation

1. Smalltalk Source Code (Smalltalk method)
2. Abstract Syntax Tree
3. Intermediate Representation
4. JavaScript Source Code
5. JavaScript evaluated: `CompiledMethod`

# Smalltalk to JavaScript Compilation



# Overview

# Representing Layers

## Overview of Alternatives

- **Class Name Prefix:** layer contains partial methods, base class name encoded in selector, e.g.: `TangentLayer»AthensHTMLPath$curveVia:to:`
- **Method Protocols**: layer contains partial methods, base class name encoded in method protocol, e.g. `TangentLayer»curveVia:to:` in protocol `AthensHTMLPath`
- **Partial Classes:** layer references partial classes, one partial class per base class contains partial methods, e.g. `TangentLayer` and `TangentLayer class»partials ^ { AthensHTMLPathTangent }`

ContextAmber uses **Partial Classes**.

# Handout only: Representing Layers

- Class Name Prefix
  - \+ Few classes: only one class per layer.
  - \+ Protocols be used to categorize methods.
- Method Protocols
  - \+ Few classes: only one class per layer.
  - \- Protocols cannot be used to categorize methods.
- Partial Classes
  - \+ Protocols can be used to categorized methods.
  - \+ Partial methods can be shared among multiple layers.
  - \- Many classes: one per base class.

# Defining Layers and Partial Classes

```
ContextAmber
    newPartialClass: #PartialClassName
    baseClass: DemoClass
    package: 'ContextAmber-Tests'
```

Figure: Declaration of partial classes

```
ContextAmber
    newLayer: #LayerName
    layerClasses: { PartialClassName }
    instanceVariableNames: ''
    package: 'ContextAmber-Tests'
```

Figure: Declaration of layers

HPI

# Handout only: Defining Layers and Partial Classes

- Layers are subclasses of `Layer`.
- Partial classes are subclasses of `PartialClass`.
- Layers/partial classes must be defined through ContextAmber API.
- Subclassing of layers is not allowed.
- Layers can have state accessible in partial classes.
- Base class and partial classes relationships are stored in methods returning the base class and the collection of partial classes, respectively.

# Layer Activation Modes

| | |
|---|---|
| Default | `Layer»activate`<br>`Layer»deactivate` |
| Object-wise | `Object»activateLayer:`<br>`Object»deactivateLayer:`<br>`Object»resetLayer:` |
| Scoped | `BlockClosure»withLayer:`<br>`BlockClosure»withoutLayer:`<br>`BlockClosure»withResetLayer:` |

**Precedence:** global (default) → scoped → object-wise

# Handout only: Layer Activation Modes

- Scoped layer deactivation statements remain on the stack because they also affect globally activated layers.
- Object-wise deactivation statements remain on the stack because they also affect globally/scoped activated layers.
- Resetting a layer removes the layer from the stack (i.e., the original state is restored as if we never interacted with that layer on that level).
- Layers are deactivated if not specified otherwise.
- No need for resetting layers globally since it is equivalent to deactivating it globally.
- Layer activation/deactivation statements are order-sensitive only within a level (i.e., activating a layer on an object and deactivating it globally afterwards does not deactivate the layer).

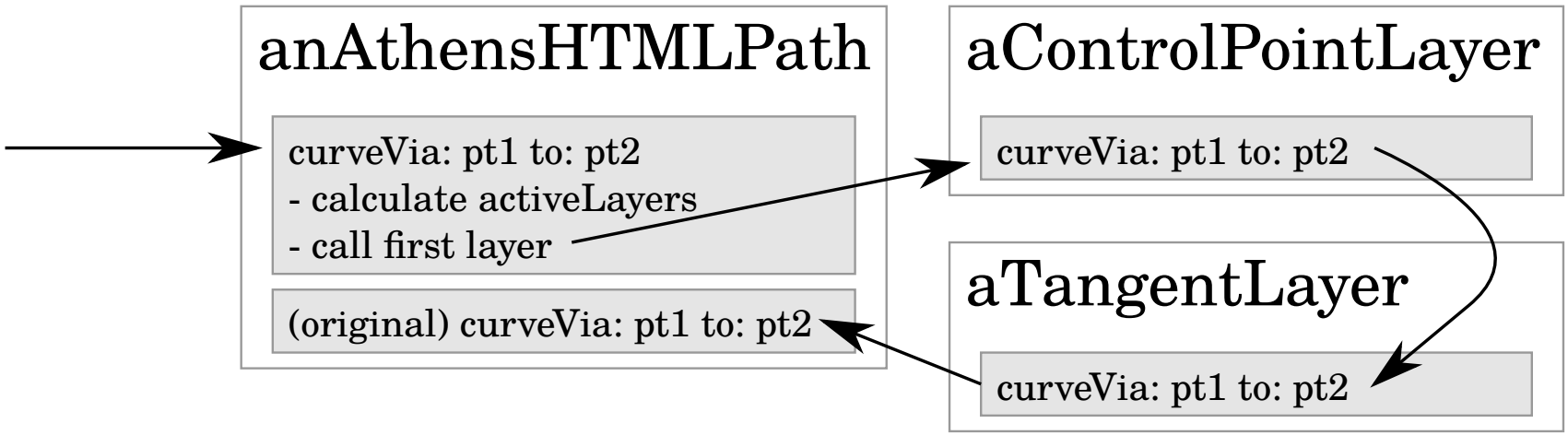
# Example: Layer Activation

| default | | object-wise | | scoped | |
|---|---|---|---|---|---|
| instr. | stack | instr. | stack | instr. | stack |
| +L1 | +L4 | +L1 | -L2 | -L1 | +L2 |
| +L2 | +L2 | +L3 | +L1 | -L1 | -L2 |
| -L1 | | +L2 | | *L1 | -L4 |
| -L3 | | *L3 | | -L4 | |
| +L4 | | -L2 | | -L2 | |
| | | | | +L2 | |

composed stack (+L2, +L4, +L1, -L2, -L4, -L2, +L2)

layer composition (L1, L2)

# Unoptimized Method Execution

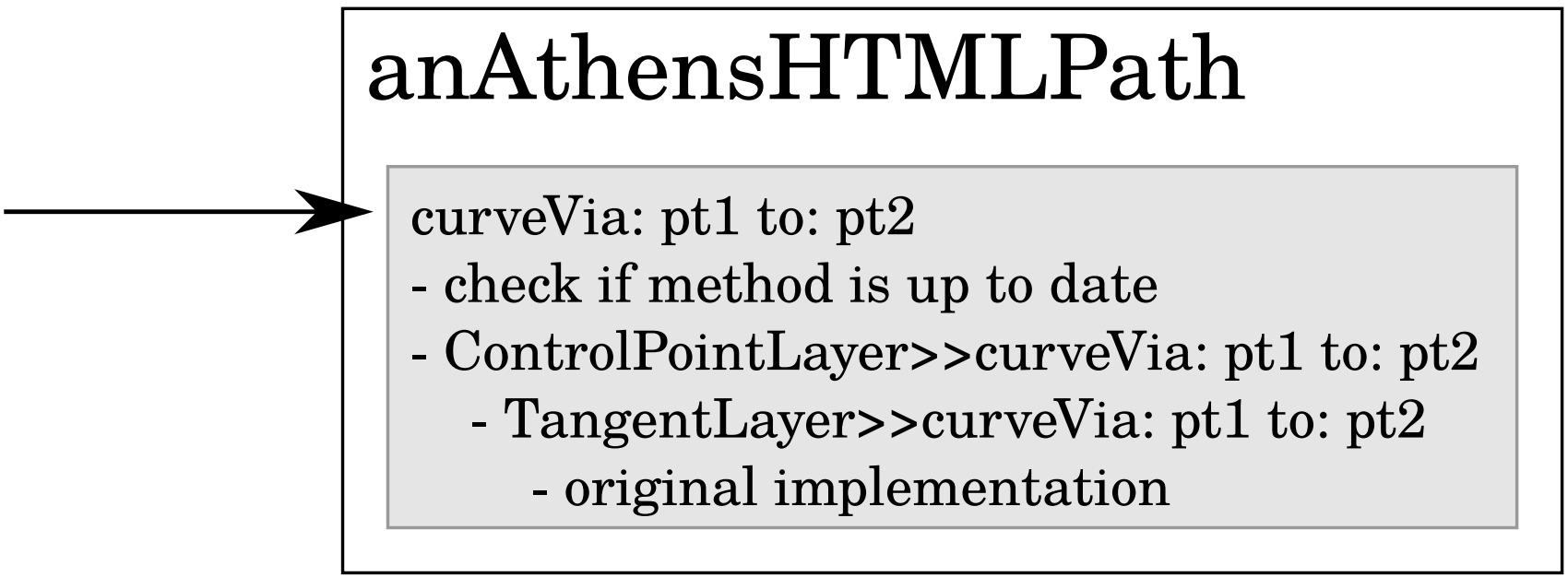



1. `C := self activeLayers.`
   Merge global stack, scoped stack, object stack.
2. `nextLayer := C detect: [ :layer | layer hasPartial: selector in: base ].`
   Find top-most layer with a matching partial method.
3. `nextMethod := (nextLayer at: base)>> selector.`
   Get `CompiledMethod` object for partial method.
4. `nextMethod fn apply: self.`
   Execute partial method in the context of `self`.

# Handout only: Unoptimized Method Execution

- `Object>>activeLayers` is expensive.
- `apply` is expensive in JavaScript.
- One additional method invocation per proceed call.

# Optimized Method Inlining



# Method Inlining

- Provide one composition-specific method containing all `proceed` calls
- AST Visitor replaces `proceed` send nodes
- Replace with send node executing next partial method as closure (block)

```
DemoLayer>>method: arg
    ^ self proceed: arg + 1

DemoClass>>method: arg
    ^ 7 * arg

InlinedObject>>method: arg
    ^ ([ :arg1 | ^ 7 * arg1 ] value: arg) + 1
```

- Inlined method calls become block nodes (`[ ... ] value`)
- Returns inside method blocks must be treated as local returns instead of non-local returns

HPI

# Handout only: Method Inlining (Code)

- `ASTProceedInliner>>inlinedMethod`
  1. Get AST of top-most partial method.
  2. Visit AST, replacing `proceed` send nodes with `inlinedSend`.
  3. *Usual* compilation process (semantic analyzer, IR generation, JavaScript code generation).
- `ASTProceedInliner>>inlinedSend`: if we encounter `proceed` send node, replace send with cached `inlinedSend`, or if it does not exist yet, create it as follows
  1. `nextLayer := C detect: [ :layer | layer hasPartial: selector in: base ].`
  2. `nextMethod := (nextLayer at: base)>> selector.`
  3. `nextAST := nextMethod ast.`
  4. `(ASTProceedInliner for: selector in: base withLayers: (C until: nextLayer))visit: nextAST.`
  5. Create `BlockNode` with `nextAST sequenceNode` as block sequence node

# Object-wise Method Inlining

## Overview of Alternatives

- **Class-wide Method Inlining:** inlined methods stored in prototypes
- **Object-wide Method Inlining:** inlined methods stored in objects
- **Class-wide Wrappers with Method Dictionaries:** one dictionary per `CompiledMethod` stores mapping from layer composition to inlined method
- **Cached Class-wide/Object-wide Method Inlining:** inlined methods cached in dictionary
- **Hybrid Approach:** use Cached Object-wide Method Inlining and switch to Cached Class-wide Method Inlining or Class-wide Wrappers if a lot of instances are created

ContextAmber uses the **Hybrid Approach** with caching.

# Handout only: Object-wise Method Inlining

- Class-wide Method Inlining: causes method invalidation and recompilation whenever a method is called on a different object with a different layer composition.
- Object-wide Method Inlining: not practical for a large number of instances.
- Class-wide Wrappers with Method Dictionaries: requires usage of JavaScript `apply()` / Smalltalk `perform:`. Slower than normal method invocation.
- Cached Class-wide Method Inlining: causes method invalidation whenever a method is called on a different object with a different layer composition, but probably no recompilation.
- Cached Object-wide Method Inlining: can speed up first method execution after layer composition change.

# Invalidation Causes

Why does an inlined method become outdated?

- Base method recompilation
- Partial method recompilation
- **Layer composition change**

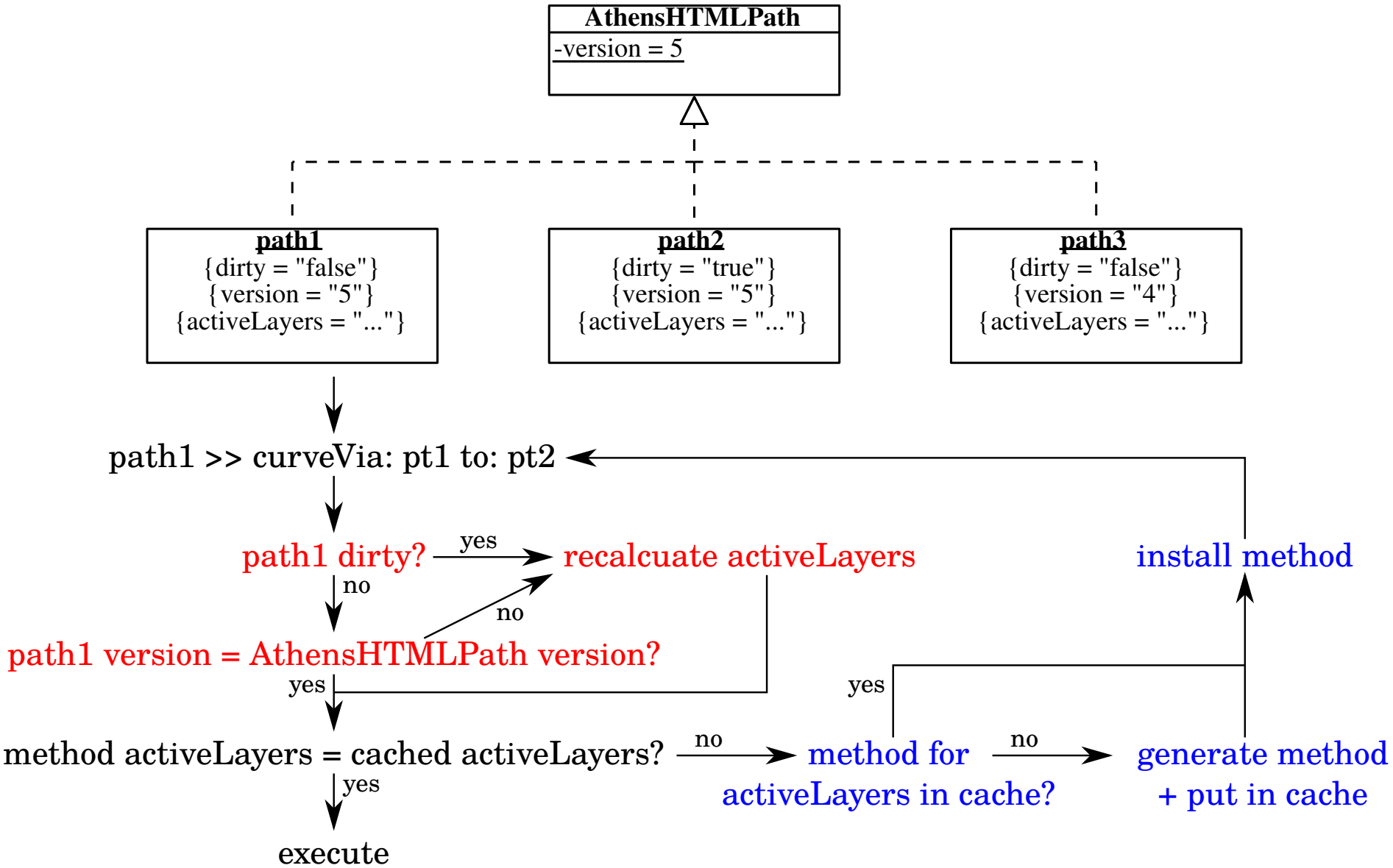
# Reinlining Point of Time

## Overview of Alternatives

When should ContextAmber reinline an outdated inlined method?

- **On Layer Composition Change:** does not work for class-wide method inlining
- **On Method Invocation:** inlined method stores its own layer composition and compares against `self activeLayers` on invocation
- **On Method Invocation with Cached** `self activeLayers`**:** update cached layer composition if global/scoped version number changed or object is dirty

# On Method Invocation with Cached `self activeLayers`



# Handout only: On Method Invocation with Cached `self activeLayers`

- `dirty` bit: indicates whether the layer composition changed for the object.
- `version` number: indicates whether the layer composition changed for all objects (scoped/global). Without the version number, ContextAmber would have to set the dirty bit on all instances.
- Red items: actions performed instead of `self activeLayers`.
- Blue items: method inlining.

# On Method Invocation with Cached `self activeLayers`

```
Object>>activeLayers
    self dirty | (self cachedGlobalVersion ~= self class
        globalVersion) | activeLayers isNil
        ifTrue: [
            activeLayers := self calculateActiveLayers.
            self dirty: false.
            self cachedGlobalVersion: self class globalVersion ].
    ^ activeLayers

Object>>activateLayer: aLayer
    ...
    self dirty: true.

Layer>>activate
    ...
    self partials do: [ :partial |
        partial base globalVersion: partial base globalVersion + 1
            ].
```

HPI

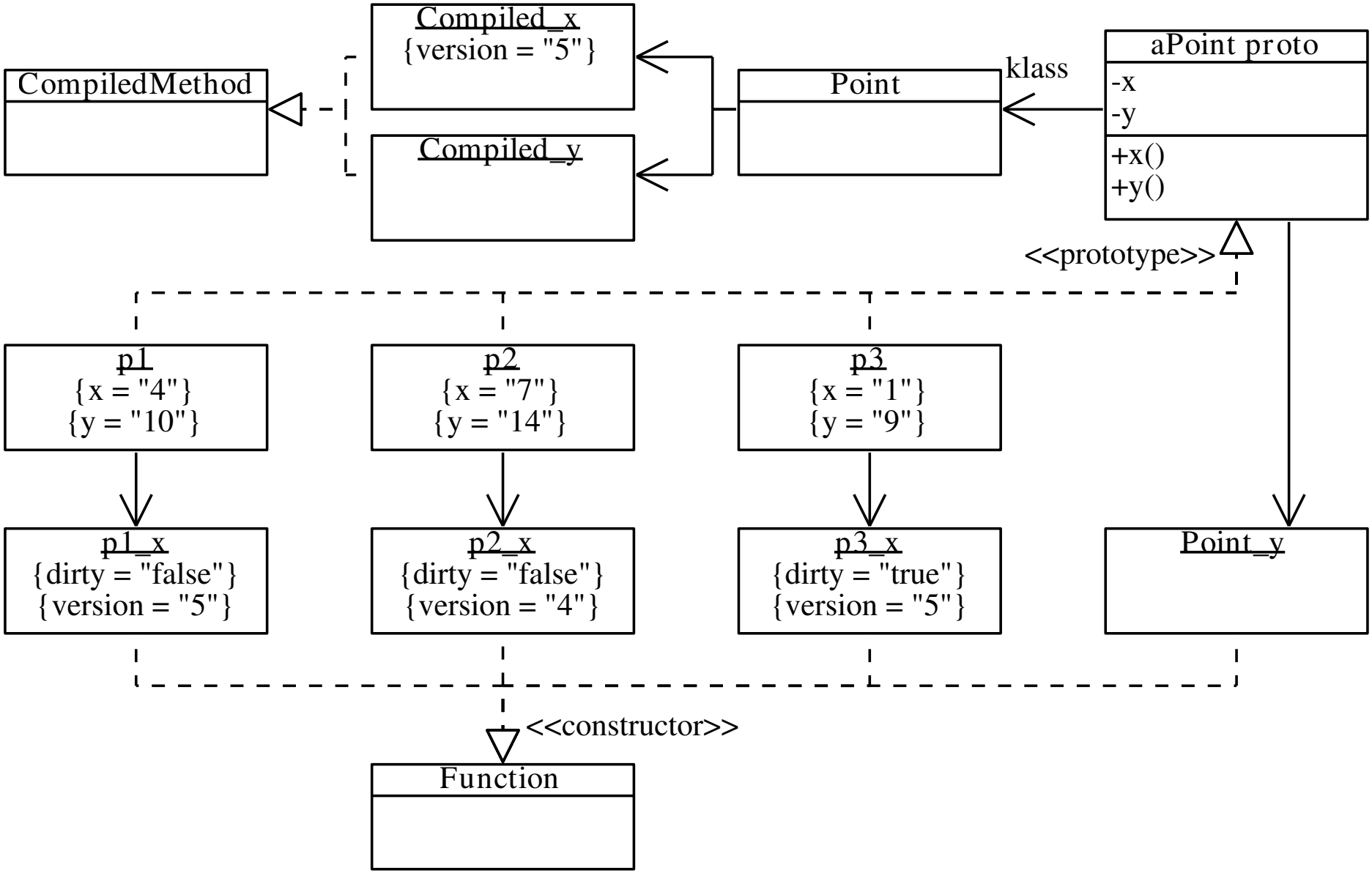
# Reinlining Point of Time

## Optimizations for Object-wide Method Inlining

**Idea:** since every object has its own method, we can immediately install a new inlined method when the version number changed or the object is dirty

- Do not compare `activeLayers` array
- Optimization not possible for Class-wide Method Inlining: even if object is not dirty and version numbers did not change, we might have to install a new inlined method (if the method is called on different objects with different layer compositions)
- Store `dirty` and `version` in method object (not practical for class-wide method inlining, because space complexity increases too much)

# Optimizations for Object-wide Method Inlining: Example



HPI

# Handout only: Optimizations for Object-wide Method Inlining: Example

- Global `version` counter per `CompiledMethod` is increased whenever the global/scoped layer compositon changes.
- `Point>>y` is inlined class-wide, `Point>>x` inlined object-wide.
- `p1 x` can be executed right away.
- `p2 x` must be reinlined because of a global/scoped layer composition change.
- `p3 x` must be reinlined because of a local/object-wise layer composition change.
- Version numbers are different from version numbers presented before! In the previous example, we used them to ensure that `activeLayers` is up to date. Here, we use them to check if an inlined method is up to date.

HPI

# Layer Signature

- Instead of version number/layer composition array, represent layer composition by unique integer
- Advoids invalidating inlined methods if a layer composition change is performed and inverted again
- Speeds up method dictionary lookup

## Current Approach

Need bijective mapping $\mathbb{L}^* \rightarrow \mathbb{Z}$

- Layer $L$ has unique ID $id(L)$
- Use Cantor's tuple function for composition $C = (L_1, L_2, \ldots, L_n)$
  $sig(C) = \pi^{n+1}(id(L_1), id(L_2), \ldots, id(L_n), n)$
- Problem: function grows too fast

# Overview

# Demonstration

Athens Paths with ContextAmber Demo

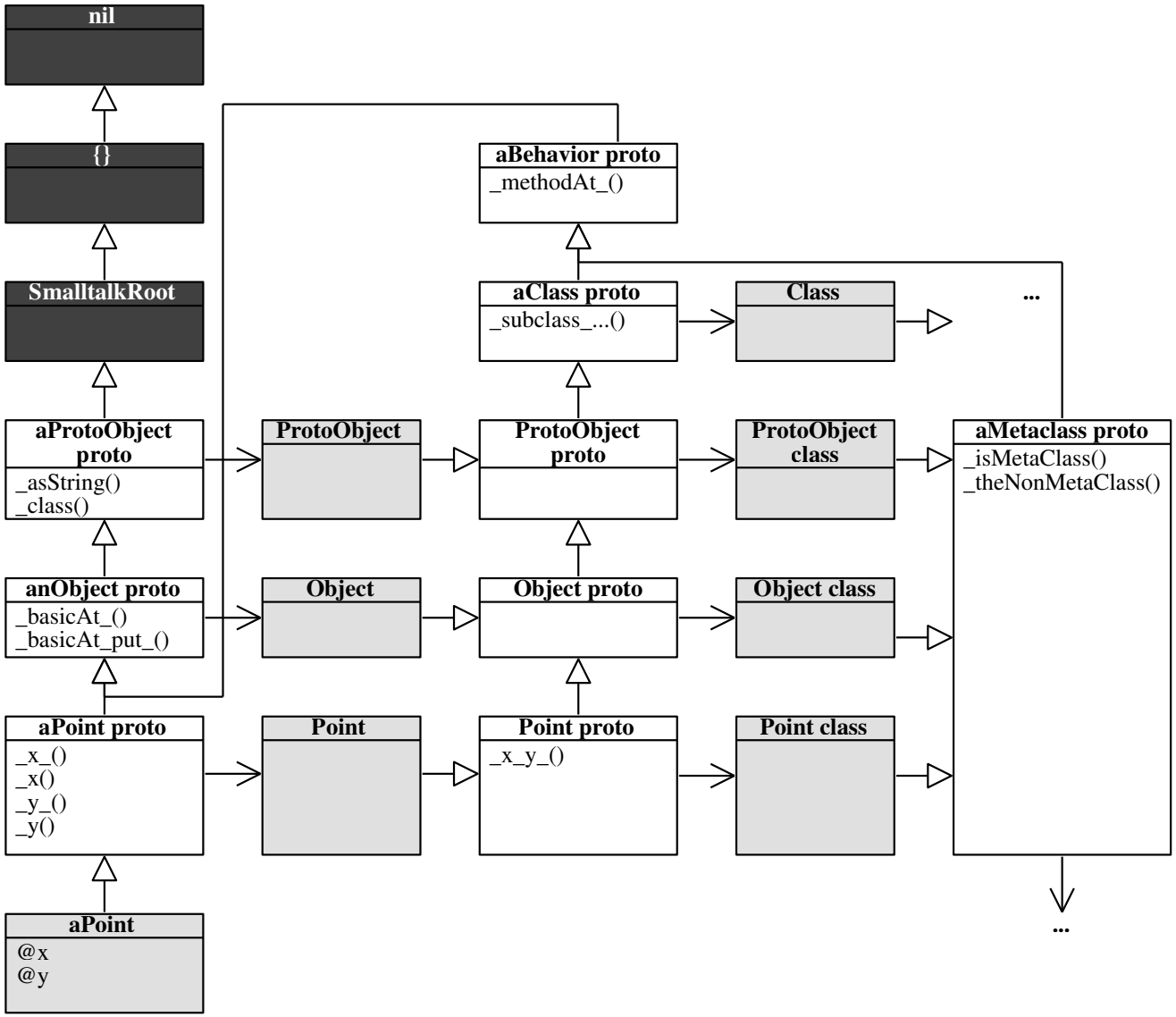
# Overview

# Project Status (work to do)

- Object-wide Method Inlining
- Class-wide Wrappers with Method Dictionaries
- Heuristics (for Hybrid Approach)
- Layer Signatures
- Exception handling: scoped layer deactivation
- IDE support for Helios
- JavaScript engine / JIT optimizations (e.g. adding attributes to objects *randomly* is discouraged)
- Performance Benchmarks: which approach is fastest?

# Appendix

# Class Model


nil
{}
SmalltalkRoot
aProtoObject proto
_asString()
_class()
anObject proto
_basicAt_()
_basicAt_put_()
aPoint proto
_x_()
_x()
_y_()
_y()
aPoint
@x
@y
ProtoObject
Object
Point
aBehavior proto
_methodAt_()
aClass proto
_subclass_...()
ProtoObject proto
Object proto
Point proto
_x_y_()
Class
ProtoObject class
Object class
Point class
...
aMetaclass proto
_isMetaClass()
_theNonMetaClass()
...

# Handout only: Class Model

- Every Smalltalk object is a JavaScript object, but not vice versa.
- Classes are implemented via the JavaScript prototype chain.
- Instances of the same class share the same prototype. The prototype contains instance methods.
- Notation: instance variables prefix , method selector prefix _ and : replaced by _
- Light gray boxes: Smalltalk objects, white boxes: prototype objects (not Smalltalk objects), black boxes: special objects

# Smalltalk to JavaScript Example (simplified)

```
Point>>+ aPoint
    ^ Point
        x: self x + aPoint asPoint x
        y: self y + aPoint asPoint y
```

```
function (aPoint) {
    var self = this;
    var $1, $2, $3, $3;

    $1 = self._x(); $2 = aPoint._asPoint();
    $3 = $2._x(); $4 = $1.__plus($3);

    $5 = self._y(); $6 = aPoint._asPoint();
    $7 = $6._y(); $8 = $5.__plus($7);

    $7 = globals.Point._x_y_($4, $8);
    return $7
}
```

# References


Bert Freudenberg, Dan H.H. Ingalls, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld.
Squeakjs: A modern and practical smalltalk that runs in any browser.
In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 57–66, New York, NY, USA, 2014. ACM.


Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld.
An open implementation for context-oriented layer composition in contextjs.
*Sci. Comput. Program.*, 76(12):1194–1209, December 2011.