Lecture 14: Memory Management

Eelco Visser

CS4200 Compiler Construction
TU Delft
December 2018

TU Delft

Garbage Collection

Reference counting

- deallocate records with count 0

Mark & sweep

- mark reachable records
- sweep unmarked records

Copying collection

- copy reachable records

Generational collection

- collect only in young generations of records

Language Agnostic Memory Management?

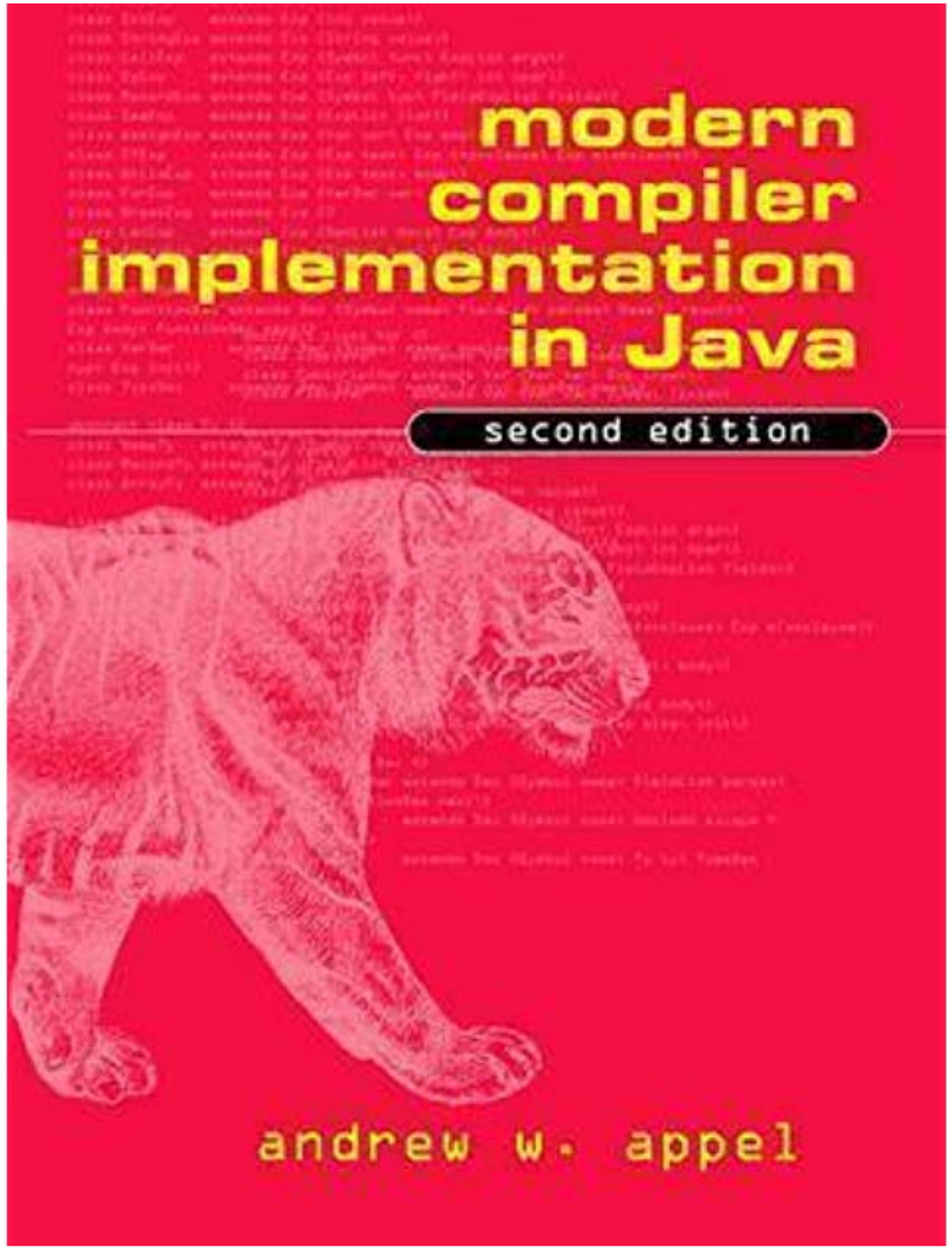
A uniform model for memory layout

- Scopes describe static binding structure
- Frames instantiate scopes at run time
- Language-parametric memory management
- Language-parametric type safety

Reading Material

Andrew W. Appel and Jens Palsberg (2002). Garbage Collection. In Modern Compiler Implementation in Java, 2nd edition. Cambridge University Press.

The lecture closely follows the discussion of mark-and-sweep collection, reference counts, copying collection, and generational collection in this chapter. This chapter also provides detailed cost analyses and discusses advantages and disadvantages of the different approaches to garbage collection.



Traditionally, operational semantics specifications use ad hoc mechanisms for representing the binding structures of programming languages.

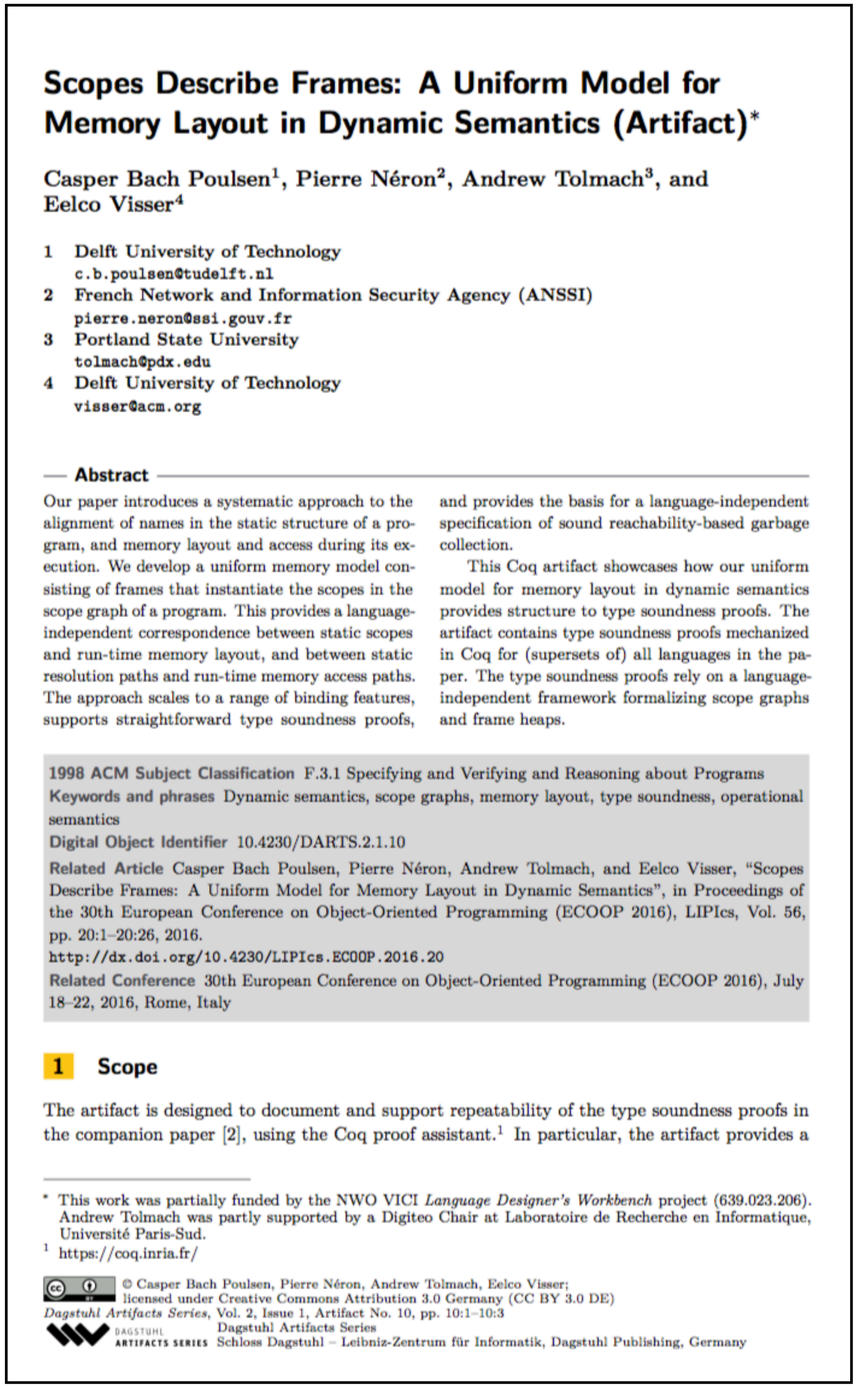
This paper introduces frames as the dynamic counterpart of scopes in scope graphs.

This provides a uniform model for the representation of memory at run-time.

We are currently experimenting with specializing DynSem interpreters using scopes and frames using Truffle/Graal with encouraging results (200x speed-ups).

ECOOP 2016

http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.20



Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics (Artifact)*

Casper Bach Poulsen[1], Pierre Néron[2], Andrew Tolmach[3], and Eelco Visser[4]

1 Delft University of Technology
c.b.poulsen@tudelft.nl
2 French Network and Information Security Agency (ANSSI)
pierre.neron@ssi.gouv.fr
3 Portland State University
tolmach@pdx.edu
4 Delft University of Technology
visser@acm.org

Abstract

Our paper introduces a systematic approach to the alignment of names in the static structure of a program, and memory layout and access during its execution. We develop a uniform memory model consisting of frames that instantiate the scopes in the scope graph of a program. This provides a language-independent correspondence between static scopes and run-time memory layout, and between static resolution paths and run-time memory access paths. The approach scales to a range of binding features, supports straightforward type soundness proofs, and provides the basis for a language-independent specification of sound reachability-based garbage collection.

This Coq artifact showcases how our uniform model for memory layout in dynamic semantics provides structure to type soundness proofs. The artifact contains type soundness proofs mechanized in Coq for (supersets of) all languages in the paper. The type soundness proofs rely on a language-independent framework formalizing scope graphs and frame heaps.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs
Keywords and phrases Dynamic semantics, scope graphs, memory layout, type soundness, operational semantics
Digital Object Identifier 10.4230/DARTS.2.1.10
Related Article Casper Bach Poulsen, Pierre Néron, Andrew Tolmach, and Eelco Visser, "Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics", in Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP 2016), LIPIcs, Vol. 56, pp. 20:1–20:26, 2016.
http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.20
Related Conference 30th European Conference on Object-Oriented Programming (ECOOP 2016), July 18–22, 2016, Rome, Italy

1 Scope

The artifact is designed to document and support repeatability of the type soundness proofs in the companion paper [2], using the Coq proof assistant.[1] In particular, the artifact provides a

* This work was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206). Andrew Tolmach was partly supported by a Digiteo Chair at Laboratoire de Recherche en Informatique, Université Paris-Sud.
[1] https://coq.inria.fr/

Memory Safety & Memory Management

Memory Safety

A program execution is memory safe if

- It only creates valid pointers through standard means
- Only uses a pointer to access memory that belongs to that pointer

Combines temporal safety and spatial safety

Spatial Safety

Access only to memory that pointer owns

View pointer as triple (p, b, e)

- p is the actual pointer
- b is the base of the memory region it may access
- e is the extent (bounds of that region)

Access allowed iff

- b <= p <= e - sizeof(typeof(p))

Allowed operations

- Pointer arithmetic increments p, leaves b and e alone
- Using &: e determined by size of original type

Temporal Safety

No access to undefined memory

Temporal safety violation: trying to access undefined memory

- Spatial safety assures it was to a legal region
- Temporal safety assures that region is still in play

Memory region is defined or undefined

Undefined memory is

- unallocated
- uninitialized
- deallocated (dangling pointers)

Memory Management

Manual memory management

- malloc, free in C
- Easy to accidentally free memory that is still in use
- Pointer arithmetic is unsafe

Automated memory management

- Spatial safety: references are opaque (no pointer arithmetic)
- (+ array bounds checking)
- Temporal safety: no dangling pointers (only free unreachable memory)

Garbage Collector

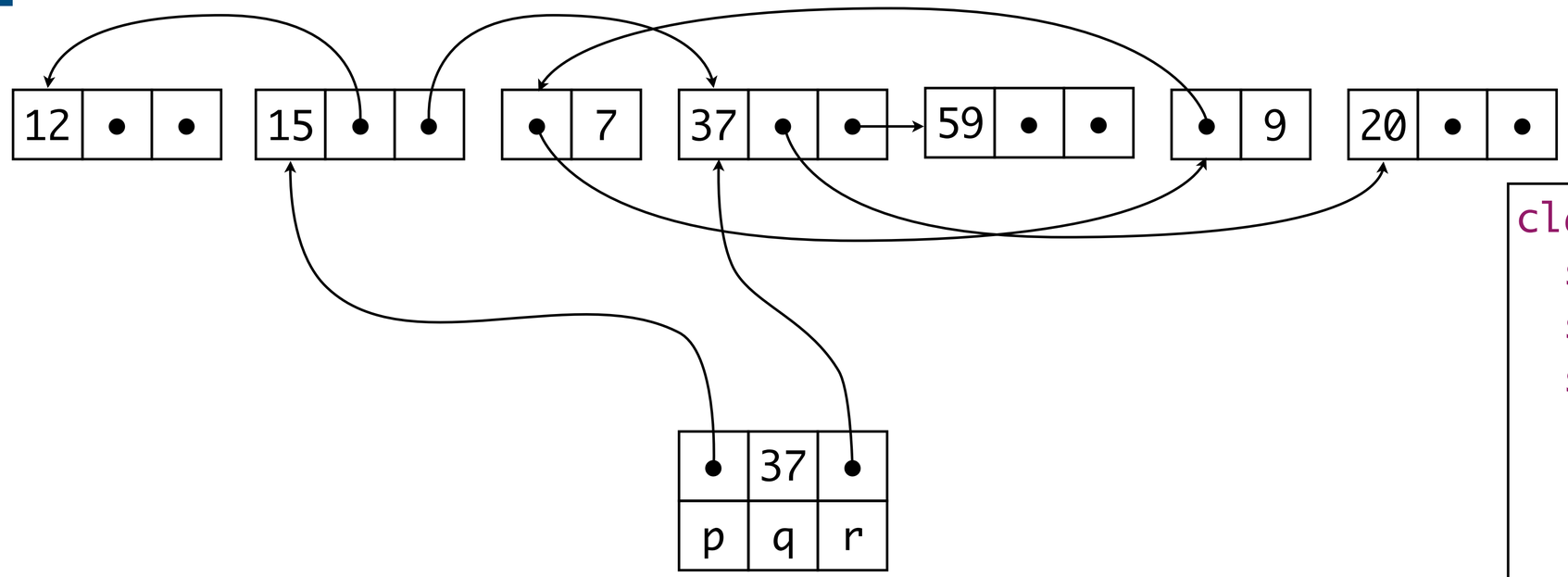
Terminology

- objects that are referenced are live
- objects that are not referenced are dead (garbage)
- objects are allocated on the heap

Responsibilities

- allocating memory
- ensuring live objects remain in memory
- garbage collection: recovering memory from dead objects

An Example Program




```
class List {
  List link;
  int key;
}

class Tree {
  int key;
  tree left;
  tree right;
}
```

```
class Main {
  static Tree makeTree() { … }
  static void showTree() { … }
  static void main() {
    {
      List x = new List(nil, 7);
      List y = new List(x, 9);
      x.link = y;
    }
    {
       Tree p = maketree();
       Tree r = p.right;
       int q = r.key;
       // garbage-collect here
       showtree(p)
    }
  }
}
```

Reference Counting

Reference Counting

Counts

- how many pointers point to each record?
- store with each record

Counting

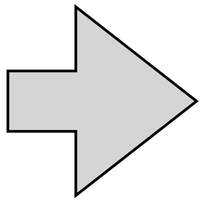
- extra instructions

Deallocate

- put on freelist
- recursive deallocation on allocation

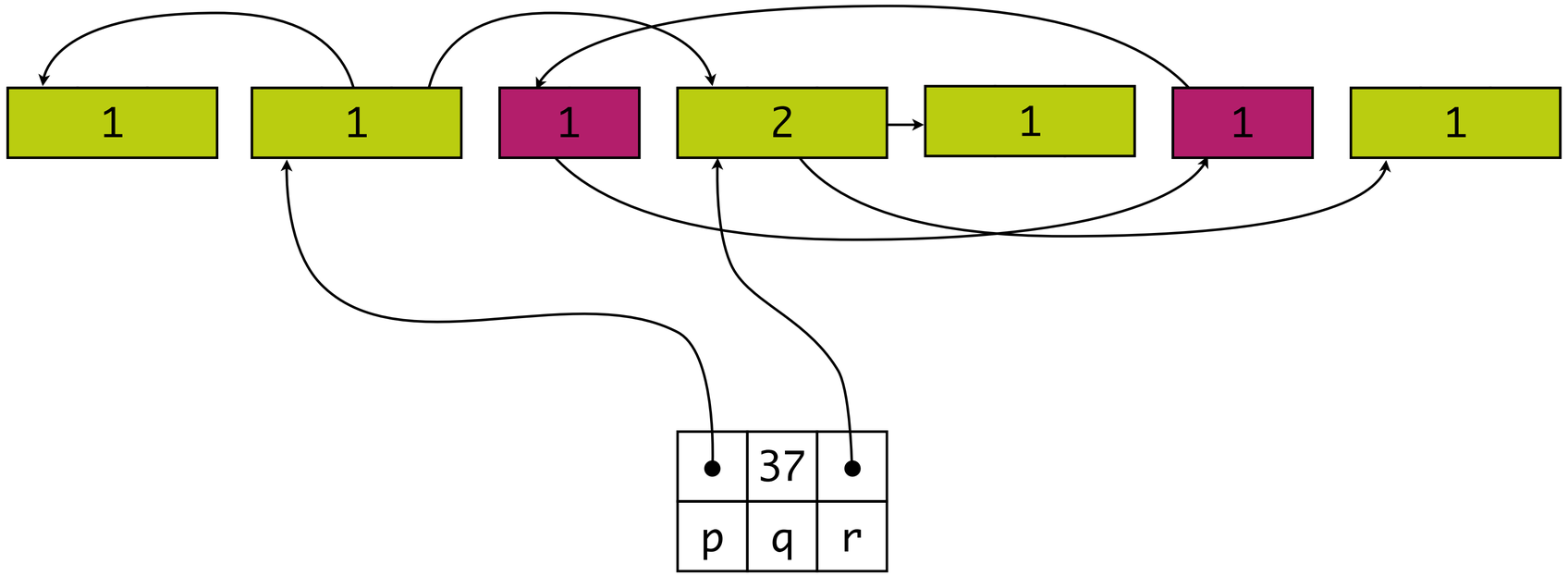
Reference Counting: Instrumentation

```
x.f := p
```



```
z       := x.f
c       := z.count
c       := c - 1
z.count := c
if (c == 0) put z on free list
x.f     := p
c       := p.count
c       := c + 1
p.count := c
```

Reference Counting



Reference Counting: Notes

Cycles

- memory leaks
- break cycles explicitly
- occasional mark & sweep collection

Expensive

- fetch, decrease, store old reference counter
- possible deallocation
- fetch, increase, store new reference counter

Programming Languages using Reference Counting

Languages with automatic reference counting

- Objective-C, Swift

Dealing with cycles

- strong reference: counts as a reference
- weak reference: can be nil, does not count
- unowned references: cannot be nil, does not count

Mark & Sweep

Mark & Sweep: Idea

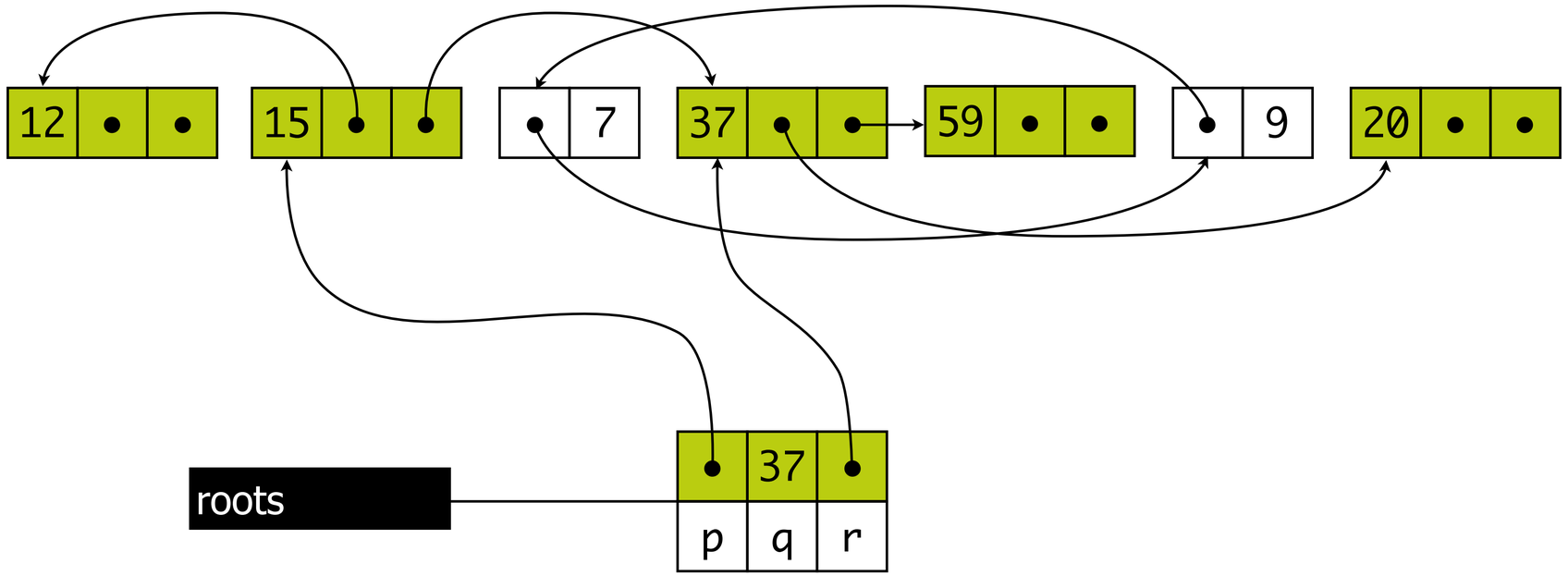
Mark

- mark reachable records
- start at variables (roots)
- follow references

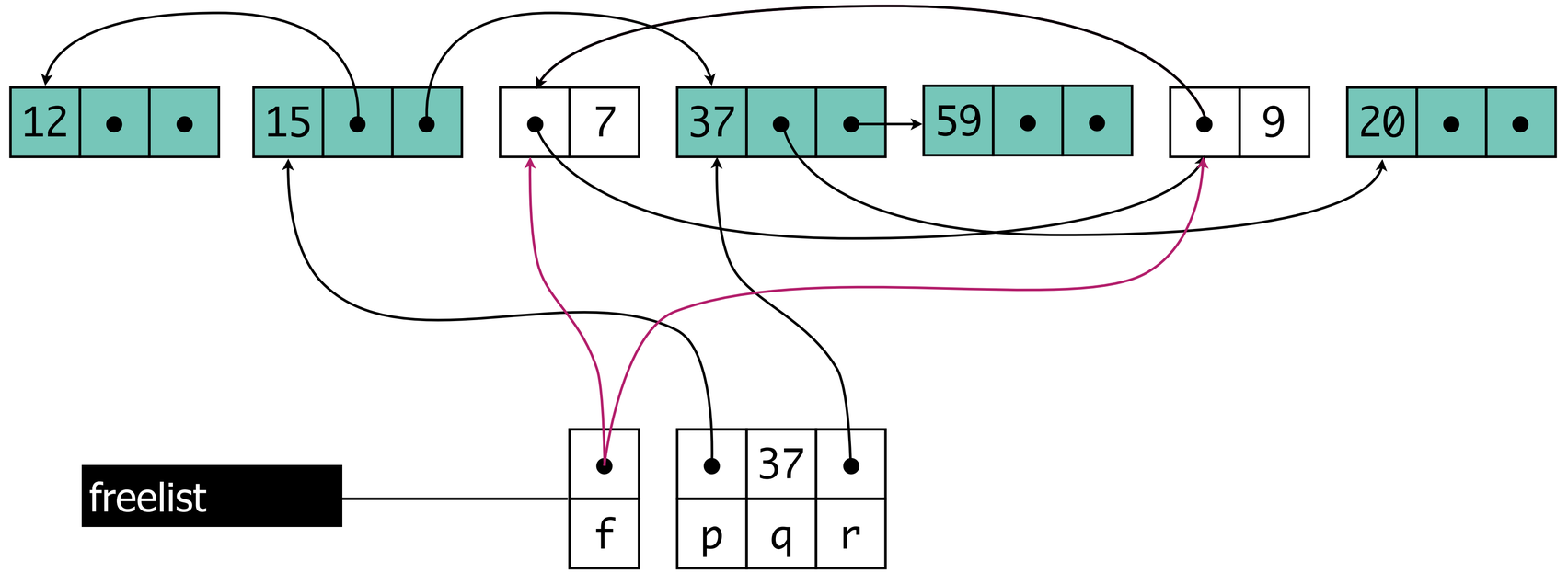
Sweep

- marked records: unmark
- unmarked records: deallocate
- linked list of free records

Marking



Sweeping



Mark & Sweep: Algorithms

```
function DFS(x)

  if pointer(x) & !x.marked

    x.marked := true

    foreach f in fields(x)

      DFS(f)
```

```
Sweep phase:

  p := first address in heap

  while p < last address in heap
    if p.marked

      p.marked := false

    else

      f1 := first field in p
      p.f1 := freelist
      free list := p

    p := p + sizeof( p )
```

Mark & Sweep: Costs

Instructions

- R reachable words in heap of size H
- Mark: c1 * R
- Sweep: c2 * H
- Reclaimed: H - R words
- Instructions per word reclaimed: (c1 * R + c2 * H) / (H - R)
- if (H >> R) cost per allocated word ~ c2

Mark & Sweep: Costs

Memory

- DFS is recursive
- maximum depth: longest path in graph of reachable data
- worst case: H
- | stack of activation records | > H

Measures

- explicit stack
- pointer reversal

Marking: DFS with Explicit Stack: Algorithms

```
function DFS(x)

  if pointer(x) & !x.marked

    x.marked = true
    t = 1 ; stack[t] = x

    while t > 0

      x = stack[t] ; t = t - 1

      foreach f in fields(x)
        if pointer(f) & !f.marked

          f.marked = true
          t = t + 1 ; stack[t] = f
```

Marking: DFS with Pointer Reversal

```
function DFS(x)
  if pointer(x) & x.done < 0
    x.done = 0 ; t = nil

    while true
     if x.done < x.fields.size
        y = x.fields[x.done]
        if pointer(y) & y.done < 0
          x.fields[x.done] = t ; t = x ; x = y ; x.done = 0
        else
          x.done = x.done + 1

      else
        y = x; x = t
        if t = nil then return
        t = x.fields[x.done]; x.fields[x.done] = y
        x.done = x.done + 1
```

marking without memory overhead

Mark & Sweep

Sweeping

- independent of marking algorithm
- several freelists (per record size)
- split free records for allocation

Fragmentation

- external: many free records of small size
- internal: too-large record with unused memory inside

Copying Collection

Copying Collection: Idea

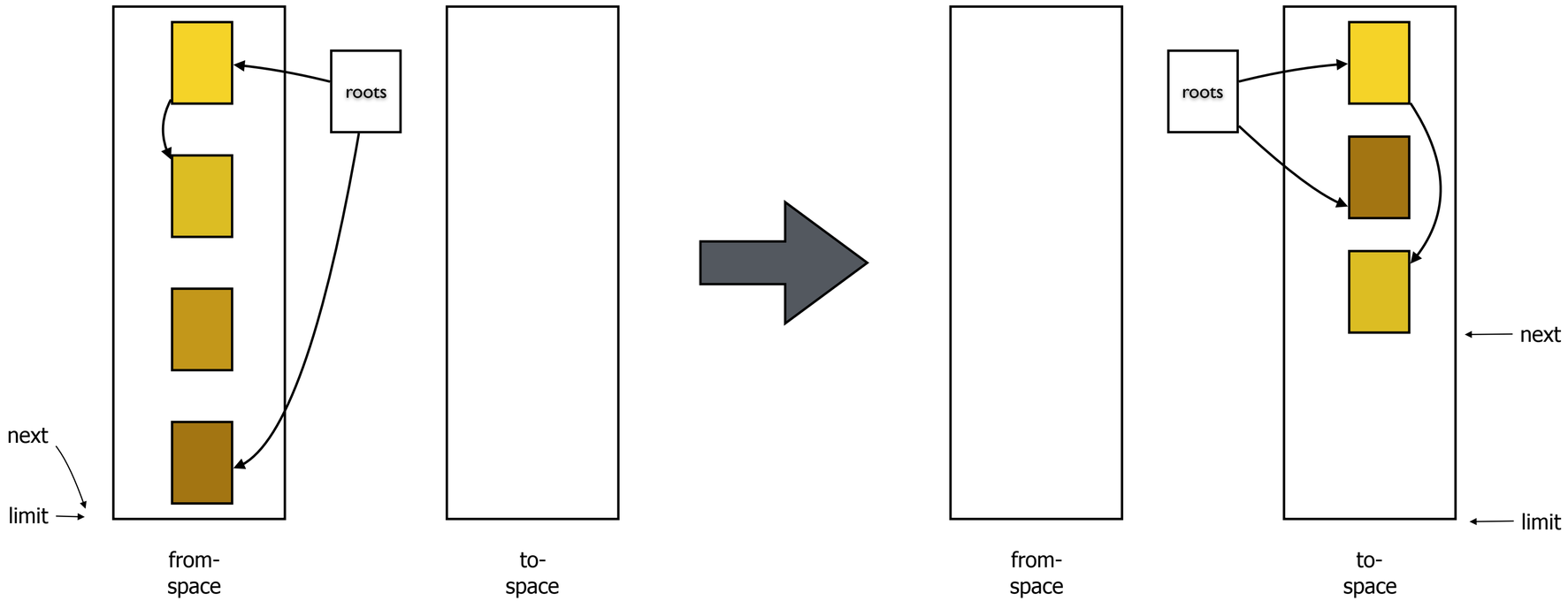
Spaces

- fromspace & tospace
- switch roles after copy

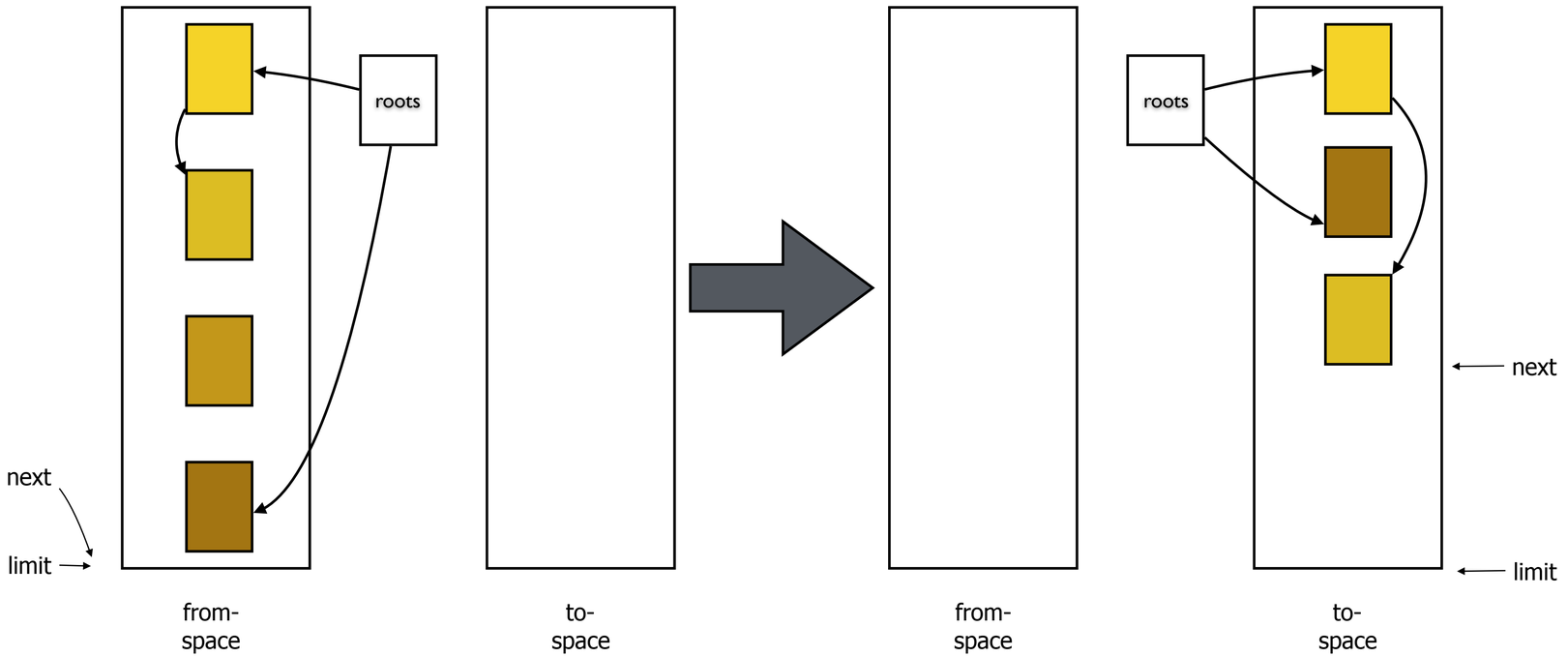
Copy

- traverse reachability graph
- copy from fromspace to tospace
- fromspace unreachable, free memory
- tospace compact, **no fragmentation**

Copying Collection: Idea

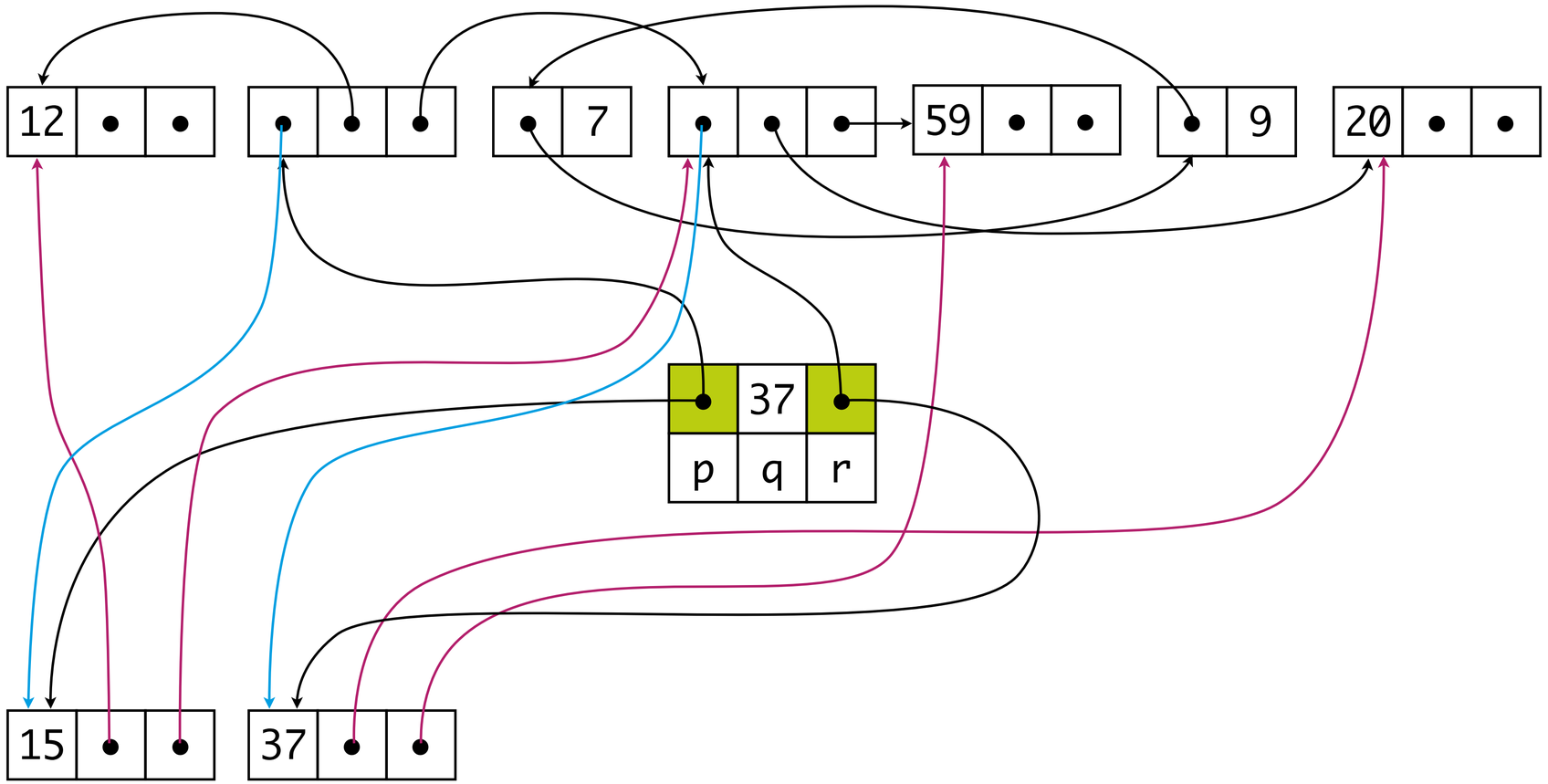


Copying Collection: Algorithm

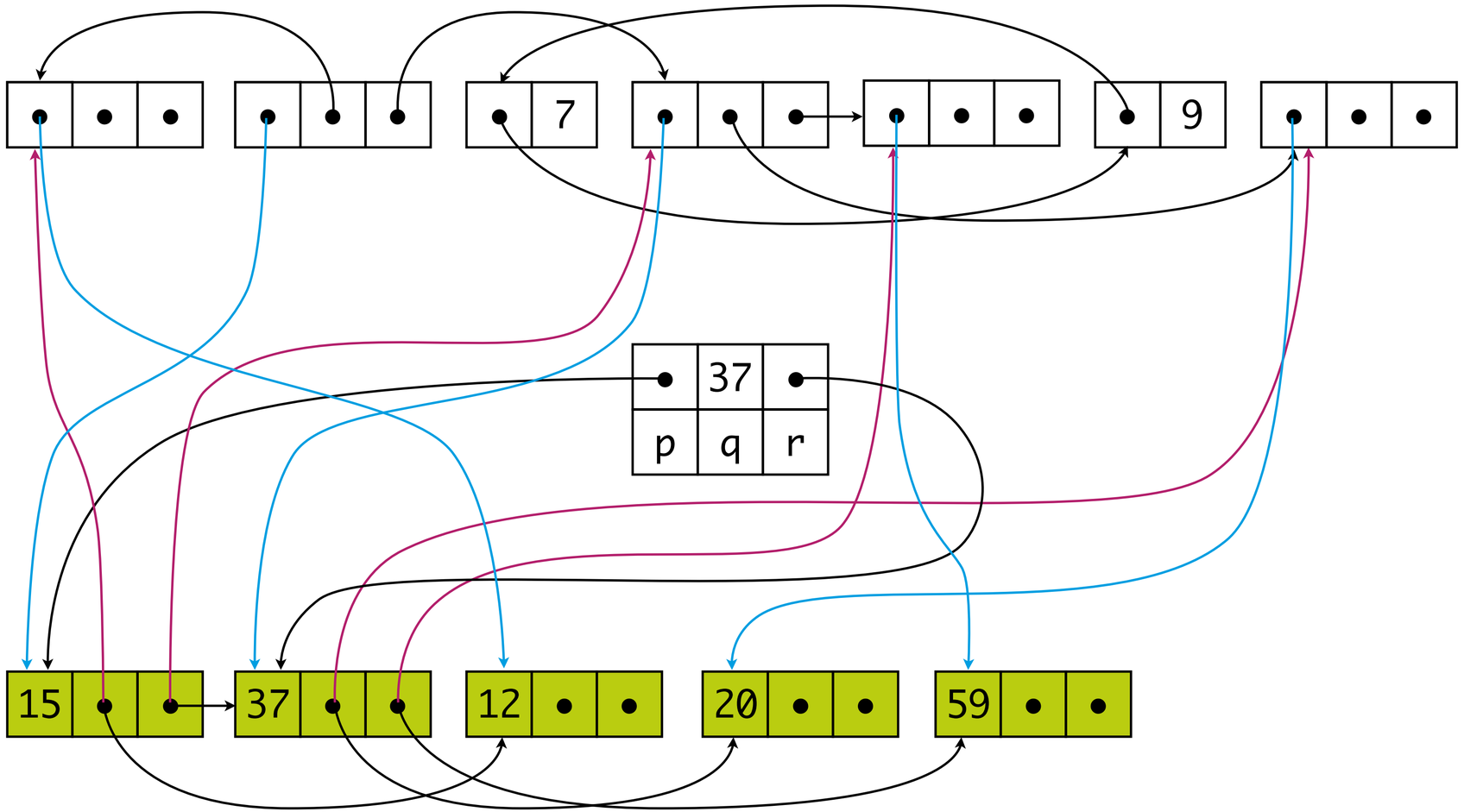



```
function BFS()

  next := scan := start(tospace)

  foreach r in roots
    r = Forward(r)

  while scan < next

    foreach f in fields of scan
      scan.f = Forward(scan.f)

    scan = scan + sizeof(scan)
```

Copying Collection: Example



Copying Collection: Example



Copying Collection: Issues

Adjacent records

- likely to be unrelated

Pointers to records in records

- likely to be accessed
- likely to be far apart

Solution

- depth-first copy: slow pointer reversals
- hybrid copy algorithm

Copying Collection: Costs

Instructions

- R reachable words in heap of size H
- BFS: c3 * R
- No sweep
- Reclaimed: H/2 - R words
- Instructions per word reclaimed: (c3 * R) / (H/2 - R)
- If (H >> R) : cost per allocated word => 0
- If (H = 4R) : c3 instructions per word allocated
- Solution: reduce portion of R to inspect => generational collection

Generational Collection

Generational Collection

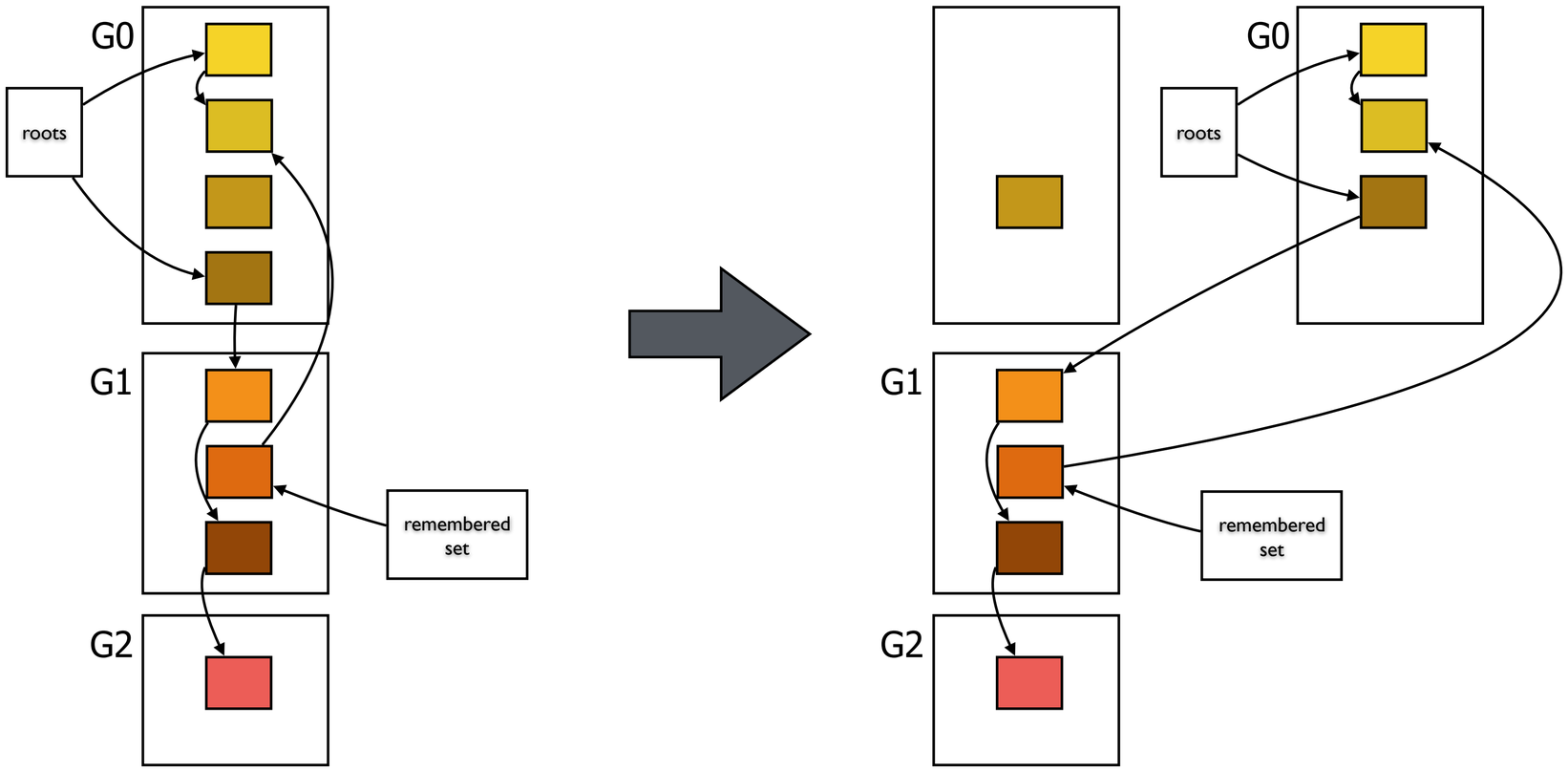
Generations

- young data: likely to die soon
- old data: likely to survive for more collections
- divide heap, collect younger generations more frequently

Collection

- roots: variables & pointers from older to younger generations
- preserve pointers to old generations
- promote objects to older generations

Generational Collection



Generational Collection: Costs

Instructions

- R reachable words in heap of size H
- BFS: c3 * R
- No sweep
- 10% of youngest generation is live: H/R = 10
- Instructions per word reclaimed:
  (c3 * R) / (H - R) = (c3 * R) / (10R - R) ~= c3/10
- Adding to remembered set: 10 instructions per update

Incremental Collection

Interrupt by garbage collector undesirable

- interactive, real-time programs

Incremental / concurrent garbage collection

- interleave collector and mutator (program)
- incremental: per request of mutator
- concurrent: in between mutator operations

Tricolor marking

- White: not visited
- Grey: visited (marked or copied), children not visited
- Black: object and children marked

Summary

Algorithms

How can we collect unreachable records on the heap?

- reference counts
- mark reachable records, sweep unreachable records
- copy reachable records

How can we reduce heap space needed for garbage collection?

- pointer-reversal
- breadth-first search
- hybrid algorithms

Design Choices

Serial vs Parallel

- garbage collection as sequential or parallel process

Concurrent vs Stop-the-World

- concurrently with application or stop application

Compacting vs Non-compacting vs Copying

- compact collected space
- free list contains non-compacted chunks
- copy live objects to new space; from-space is non-fragmented

Performance Metrics

Throughput

- percentage of time not spent in garbage collection

GC overhead

- percentage of time spent in garbage collection

Pause time

- length of time execution is stopped during garbage collection

Frequency of collection

- how often collection occurs

Footprint

- measure of (heap) size

Garbage Collection in Java HotSpot VM

Serial collector

- young generation: copying collection
- old generation: mark-sweep-compact collection

Parallel collector

- young generation: stop-the-world copying collection in parallel
- old generation: same as serial

Parallel compacting collector

- young generation: same as parallel
- old generation: roots divided in threads, marking live objects in parallel, …

Concurrent Mark-Sweep (CMS) collector

- stop-the-world initial marking and re-marking
- concurrent marking and sweeping

Literature

- Andrew W. Appel, Jens Palsberg. Modern Compiler Implementation in Java, 2nd edition, 2002.
- Sun Microsystems. Memory Management in the Java HotSpotTM Virtual Machine, April 2006.

Language-Parametric Memory Management?

Language-Parametric Memory Management?

Garbage collectors are language-specific

- Representation of objects in memory
- Roots of heap in stack

Can we derive garbage collector from language definition?

Language-Parametric Type Safety?

Type Safety: Well-typed programs don't go wrong

- A program that type checks does not have run-time type errors
- Preservation
  - `e : t & e -> v => v : t`
- Progress
  - `e -> e'  =>  e' is a value || e' -> e''`
- (Slightly different for big step semantics as in definitional interpreters)

Proving type safety

- Easier to establish with an interpreter
- Bindings complicate proof
- How to maintain?
- Can we automate verification of type safety?

Scopes Describe Frames

Traditionally, operational semantics specifications use ad hoc mechanisms for representing the binding structures of programming languages.

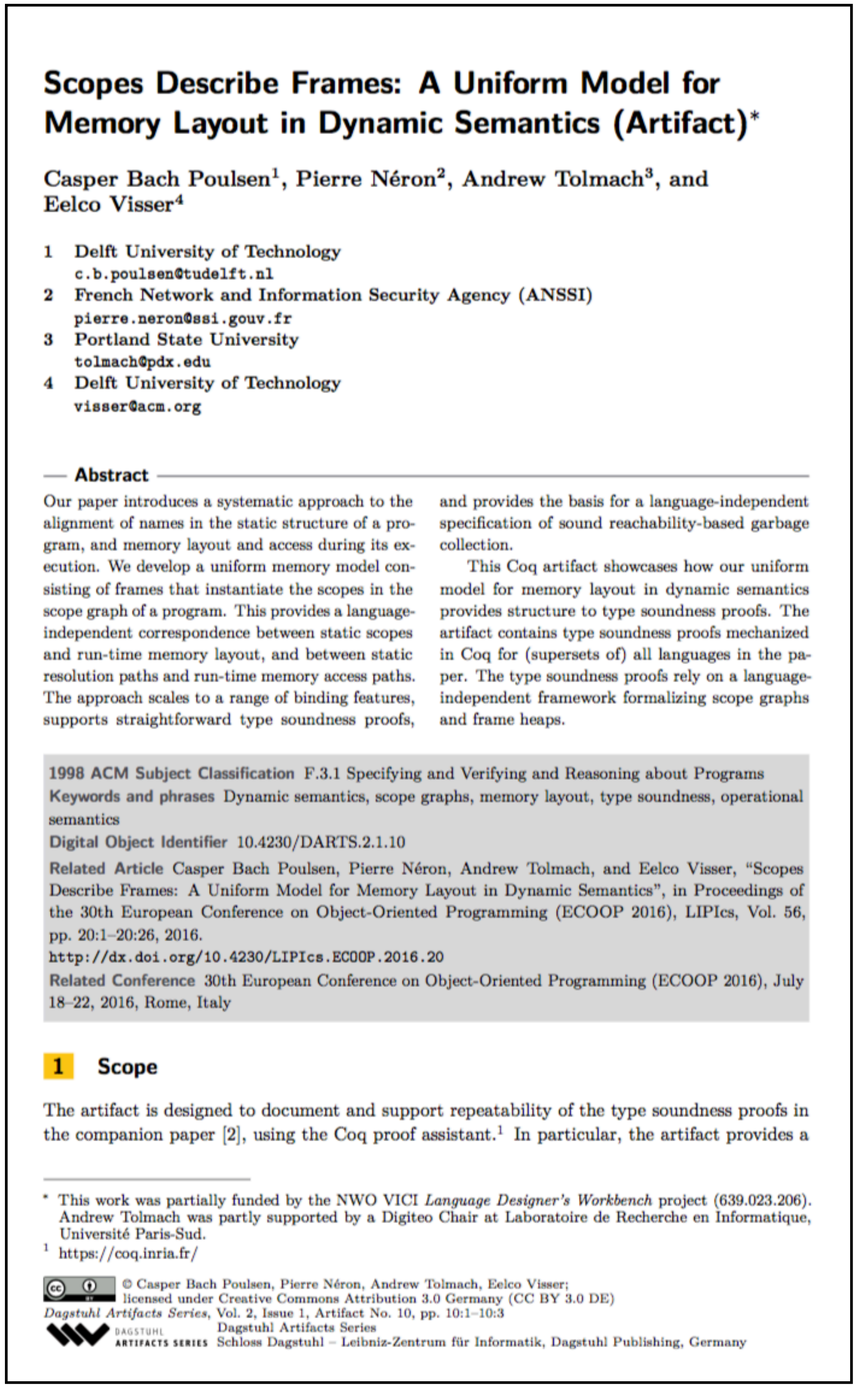
This paper introduces frames as the dynamic counterpart of scopes in scope graphs.

This provides a uniform model for the representation of memory at run-time.

We are currently experimenting with specializing DynSem interpreters using scopes and frames using Truffle/Graal with encouraging results (200x speed-ups).

ECOOP 2016

http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.20



Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics (Artifact)*

Casper Bach Poulsen[1], Pierre Néron[2], Andrew Tolmach[3], and Eelco Visser[4]

1 Delft University of Technology
c.b.poulsen@tudelft.nl
2 French Network and Information Security Agency (ANSSI)
pierre.neron@ssi.gouv.fr
3 Portland State University
tolmach@pdx.edu
4 Delft University of Technology
visser@acm.org

Abstract

Our paper introduces a systematic approach to the alignment of names in the static structure of a program, and memory layout and access during its execution. We develop a uniform memory model consisting of frames that instantiate the scopes in the scope graph of a program. This provides a language-independent correspondence between static scopes and run-time memory layout, and between static resolution paths and run-time memory access paths. The approach scales to a range of binding features, supports straightforward type soundness proofs, and provides the basis for a language-independent specification of sound reachability-based garbage collection.

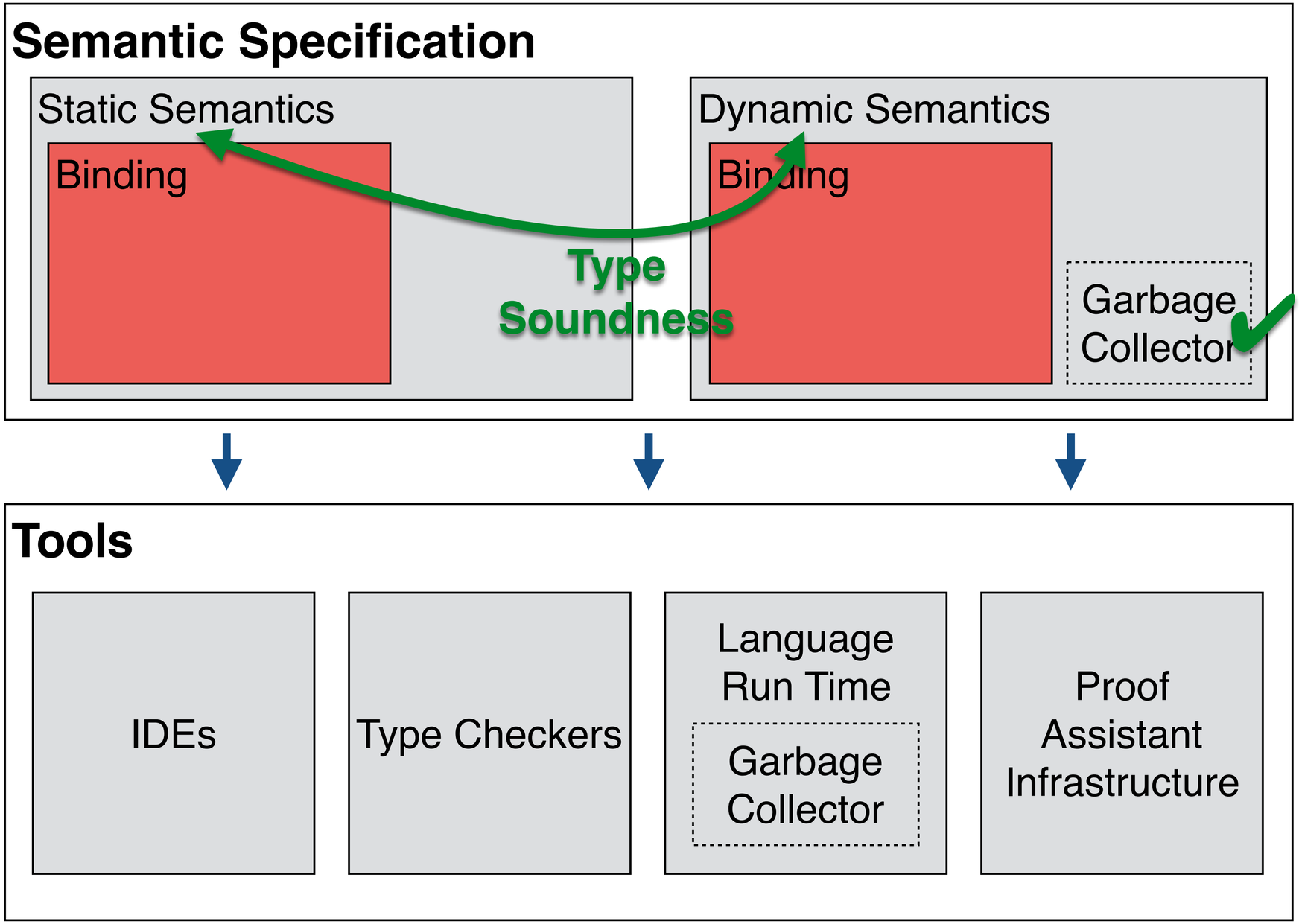
This Coq artifact showcases how our uniform model for memory layout in dynamic semantics provides structure to type soundness proofs. The artifact contains type soundness proofs mechanized in Coq for (supersets of) all languages in the paper. The type soundness proofs rely on a language-independent framework formalizing scope graphs and frame heaps.

1998 ACM Subject Classification F.3.1 Specifying and Verifying and Reasoning about Programs
Keywords and phrases Dynamic semantics, scope graphs, memory layout, type soundness, operational semantics
Digital Object Identifier 10.4230/DARTS.2.1.10
Related Article Casper Bach Poulsen, Pierre Néron, Andrew Tolmach, and Eelco Visser, "Scopes Describe Frames: A Uniform Model for Memory Layout in Dynamic Semantics", in Proceedings of the 30th European Conference on Object-Oriented Programming (ECOOP 2016), LIPIcs, Vol. 56, pp. 20:1–20:26, 2016.
http://dx.doi.org/10.4230/LIPIcs.ECOOP.2016.20
Related Conference 30th European Conference on Object-Oriented Programming (ECOOP 2016), July 18–22, 2016, Rome, Italy

1 Scope

The artifact is designed to document and support repeatability of the type soundness proofs in the companion paper [2], using the Coq proof assistant.[1] In particular, the artifact provides a

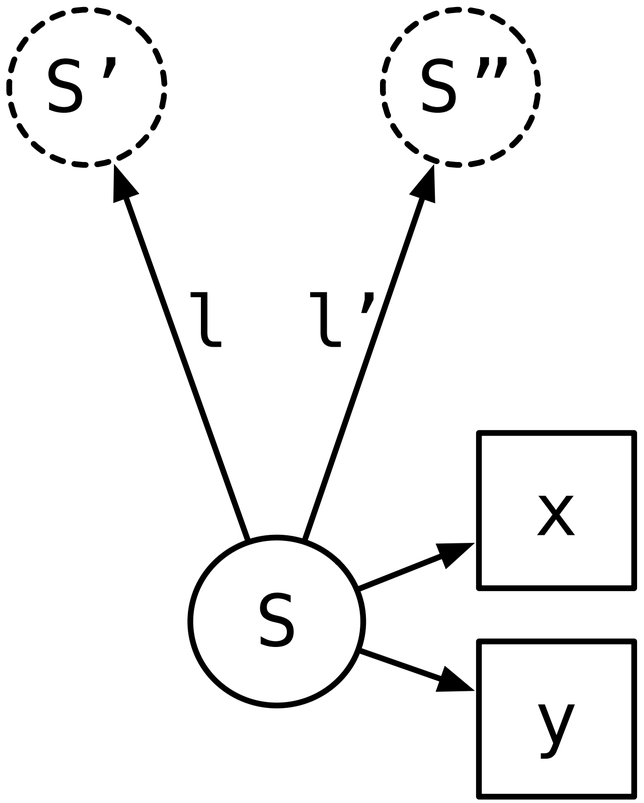
* This work was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206). Andrew Tolmach was partly supported by a Digiteo Chair at Laboratoire de Recherche en Informatique, Université Paris-Sud.
[1] https://coq.inria.fr/


Semantic Specification
Static Semantics
Binding
Dynamic Semantics
Binding
Type
Soundness
Garbage
Collector
Tools
IDEs
Type Checkers
Language
Run Time
Garbage
Collector
Proof
Assistant
Infrastructure

		Static	Dynamic
Lexical	`val x = 31;` `val y = x + 11;`	Typing Contexts Type Substitution	Substitution Environments De Bruijn Indices HOAS
Mutable	`var x = 31;` `x = x + 11;`	Typing Contexts Store Typing	Stores/Heaps

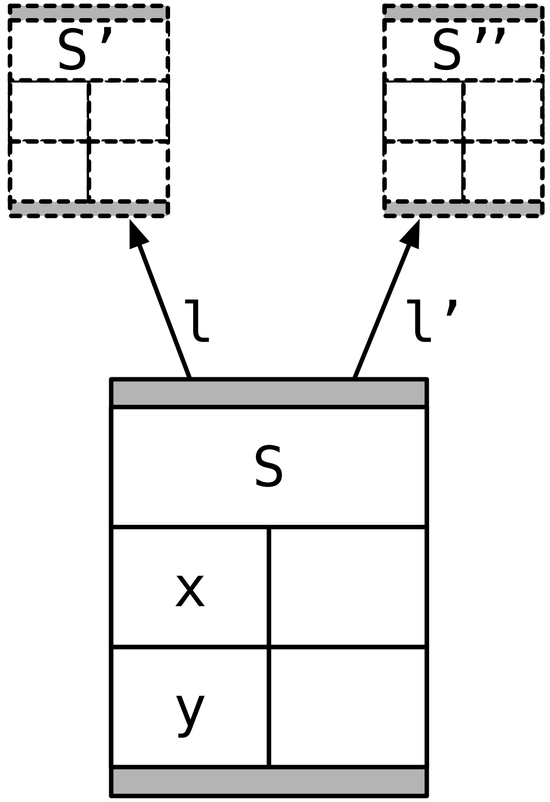
Objects	`class A {`
	`  var x = 0;`
	`  var y = 42;`
	`}`
	`var r = new A();`
		Class Tables	Mutable Objects
			Stores/Heaps

Scope



[ESOP’15]

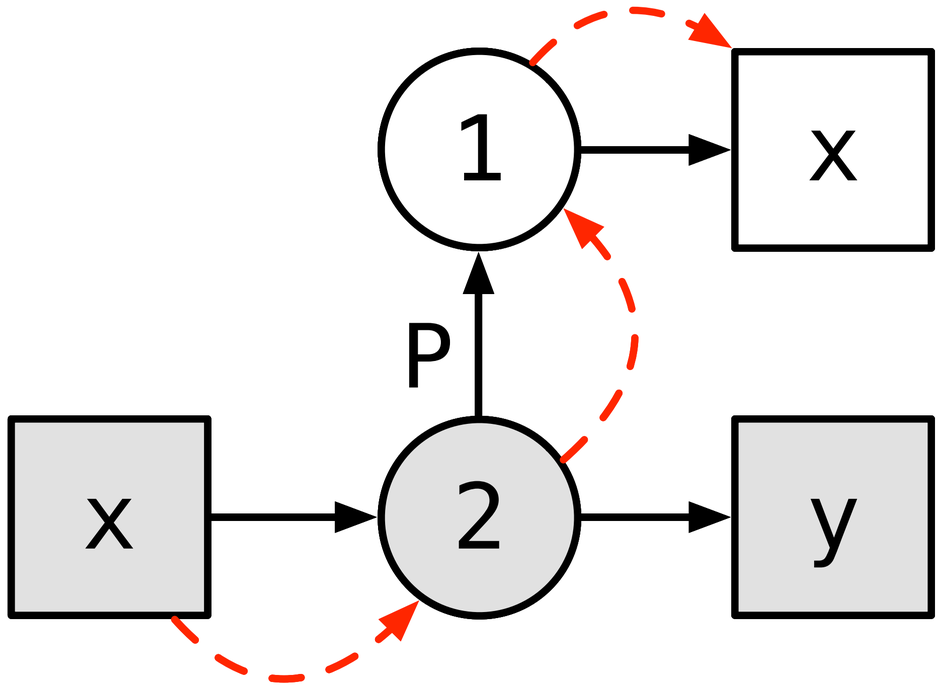
Frame



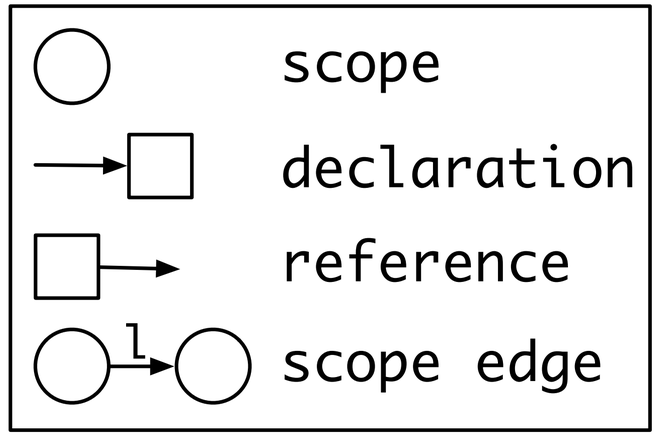
[ECOOP’16]

Lexical Scoping

```
val x = 31;
val y = x + 11;
```

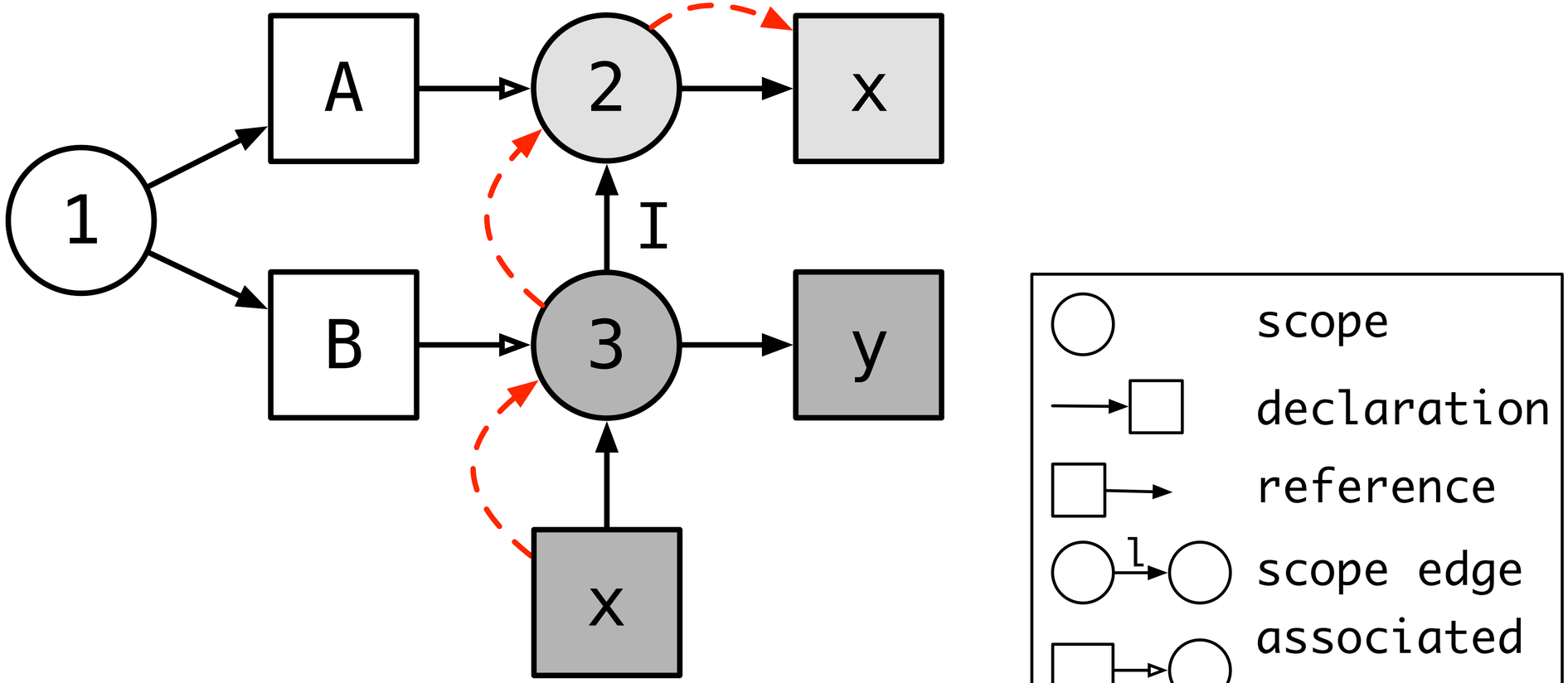


[ESOP'15; PEPM'16]



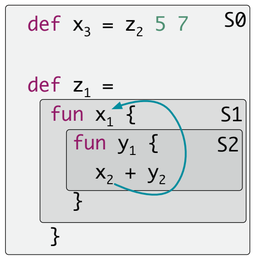
Inheritance

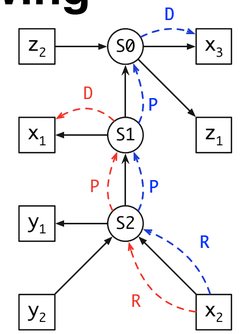
```
class A { var x = 42; }
class B extends A { var y = x; }
```



More Binding Patterns

Shadowing





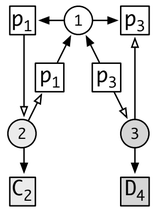
$$\frac{D < P.p \qquad p < p'}{s.p < s.p'}$$

R.P.D < R.P.P.D

Java Packages

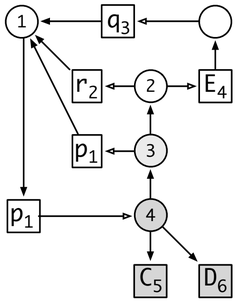
```
package p1;
  class C2 {}
```

```
package p3;
  class D4 {}
```

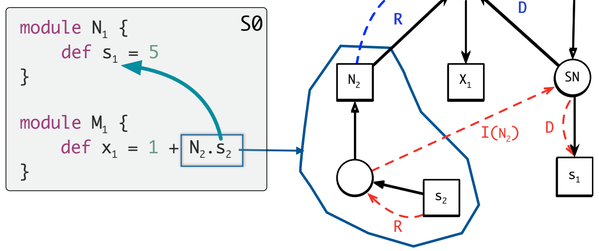


Java Import

```
package p1;

imports r2.*;
imports q3.E4;

public class C5 {}

class D6 {}
```



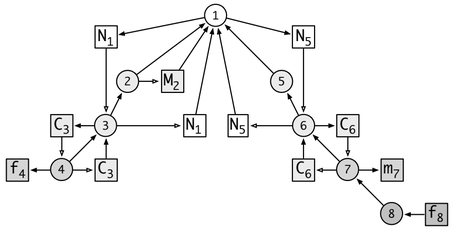
Qualified Names



C# Namespaces and Partial Classes

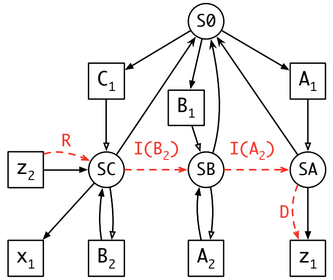
```
namespace N1 {
  using M2;
  partial class C3 {
    int f4;
  }
}
```

```
namespace N5 {
  partial class C6 {
    int m7() {
      return f8;
    }
  }
}
```



Transitive vs. Non-Transitive

```
module A1 {
  def z1 = 5        SA
}                ??
module B1 {
  import A2         SB
}
module C1 {
  import B2         SC
  def x1 = 1 + z2
}
```



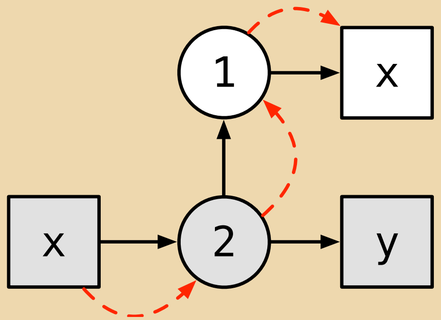
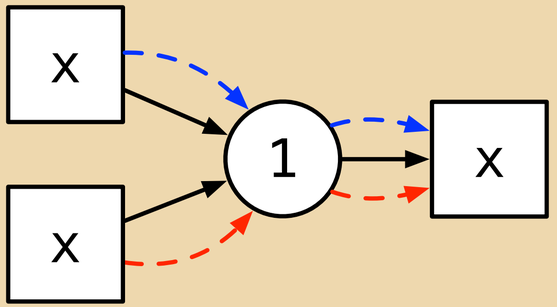
With transitive imports, a well formed path is R.P*.I(_)*.D

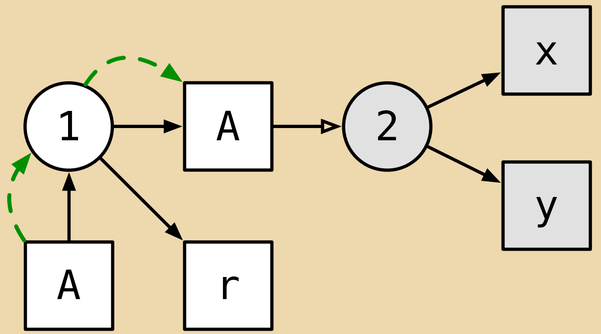
With non-transitive imports, a well formed path is R.P*.I(_)?.D

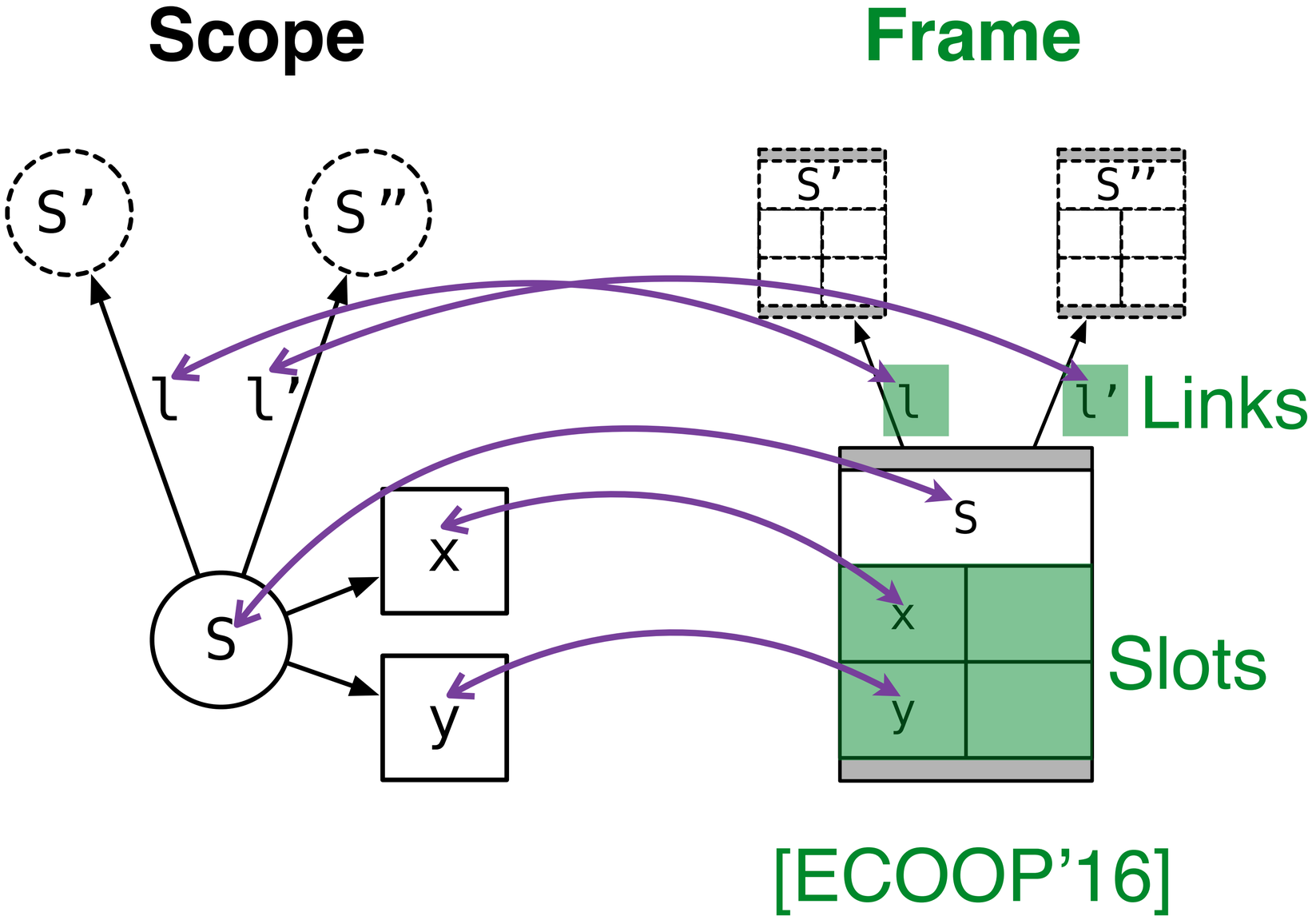
[ESOP'15]

		Static	Dynamic
Lexical	`val x = 31;` `val y = x + 11;`	Typing Contexts Type Substitution	Substitution Environments De Bruijn Indices HOAS
Mutable	`var x = 31;` `x = x + 11;`	Typing Contexts Store Typing	Stores/Heaps

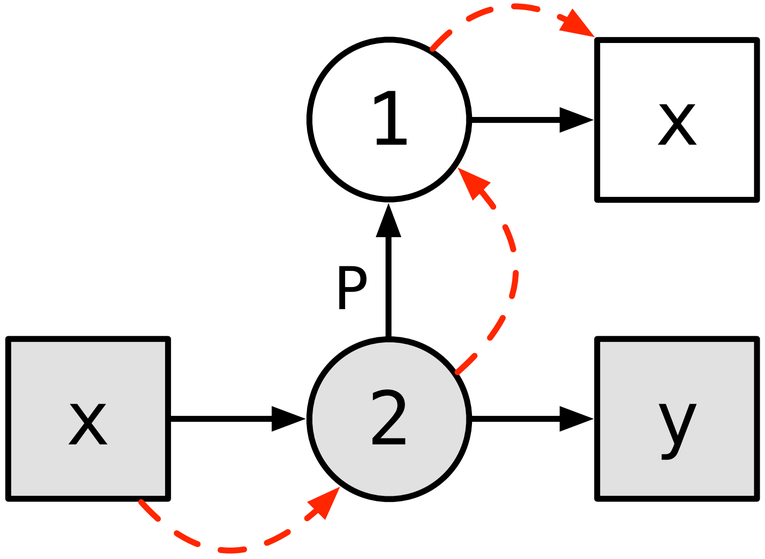
Objects	`class A {`
	`  var x = 0;`
	`  var y = 42;`
	`}`
	`var r = new A();`
		Class Tables	Mutable Objects
			Stores/Heaps

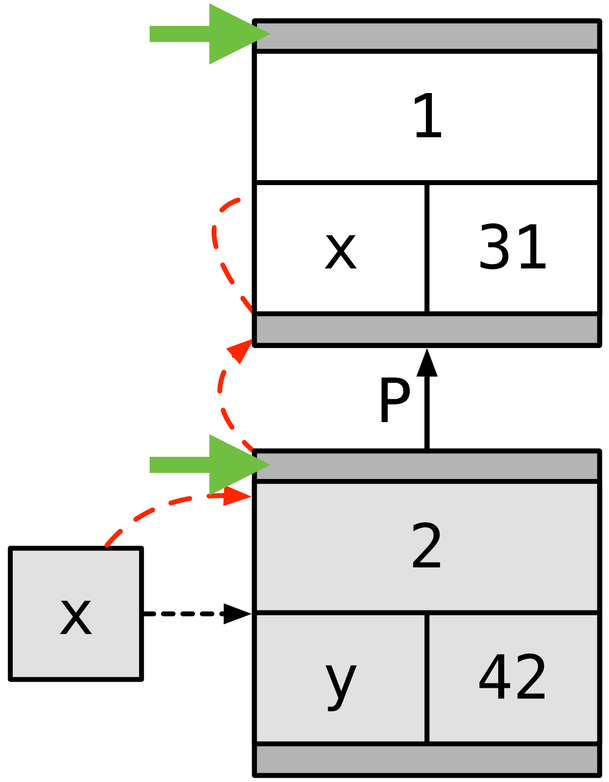
	Static	Dynamic
Lexical	`val x = 31;` `val y = x + 11;`		Substitution Environments De Bruijn Indices HOAS
Mutable	`var x = 31;` `x = x + 11;`		Stores/Heaps

Objects	`class A {`
	`  var x = 0;`
	`  var y = 42;`
	`}`
	`var r = new A();`
			Mutable Objects
			Stores/Heaps

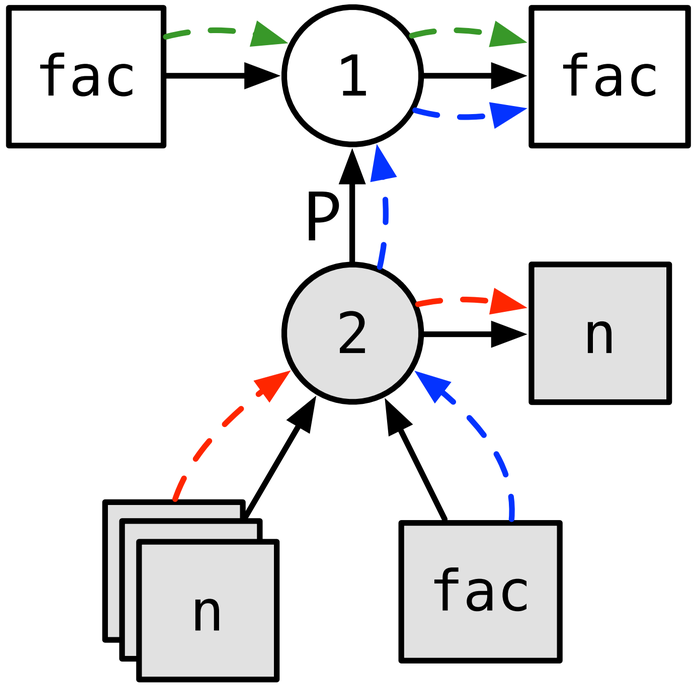

Scope
Frame
S’
S”
l
l’
x
y
S
S’
S”
l
l’
Links
s
x
y
Slots
[ECOOP’16]

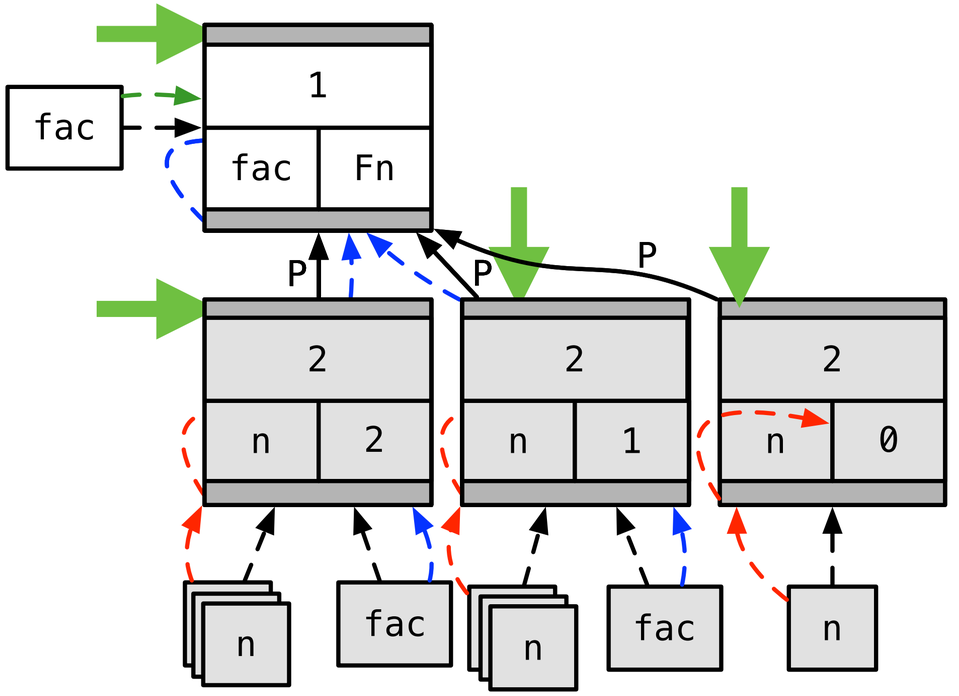
```
val x = 31;
val y = x + 11;
```



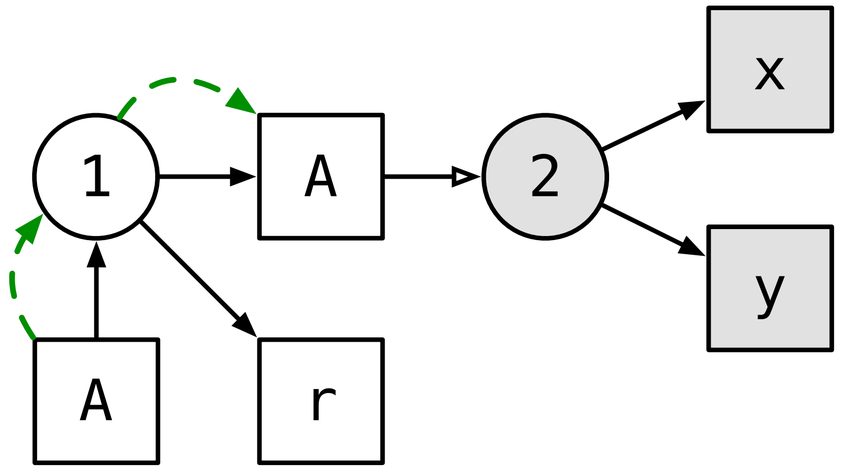


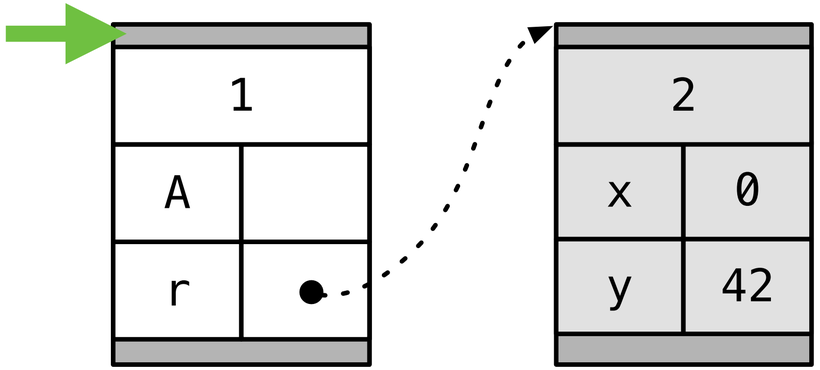
```
def fac(n : Int) : Int = {
  if (n == 0) 1
  else n * fac(n - 1)
};
fac(2);
```

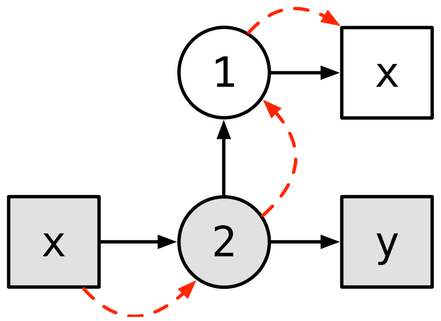
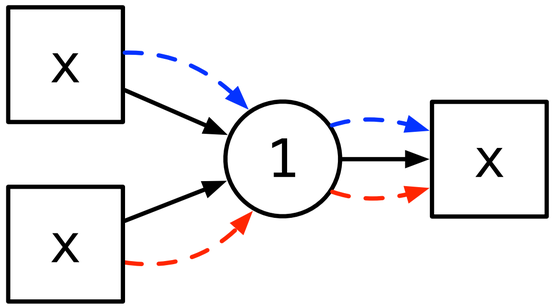


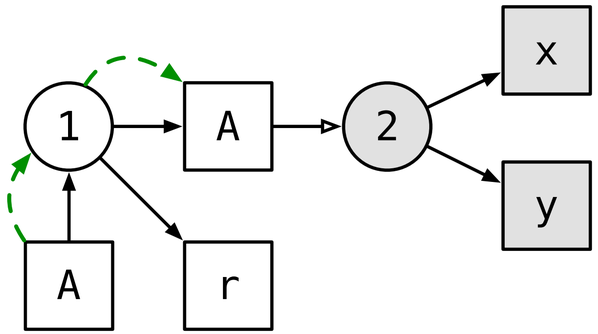


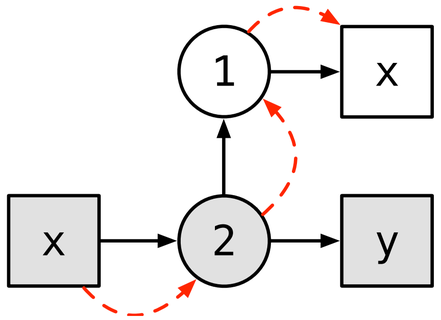
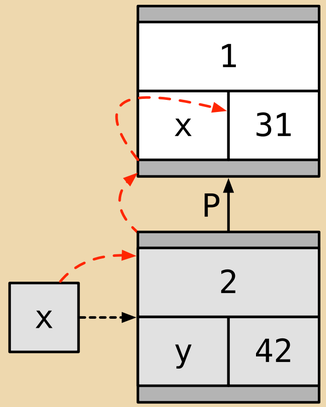
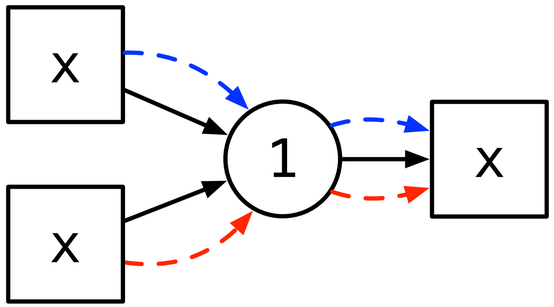
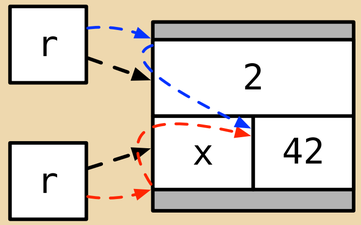
```
class A {
          var x = 0;
          var y = 42; }
var r = new A();
```

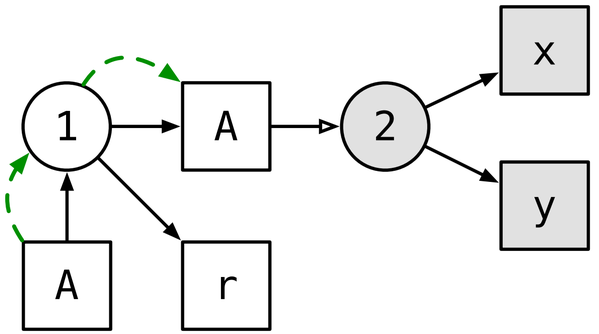
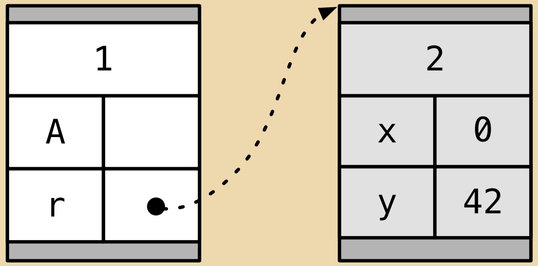




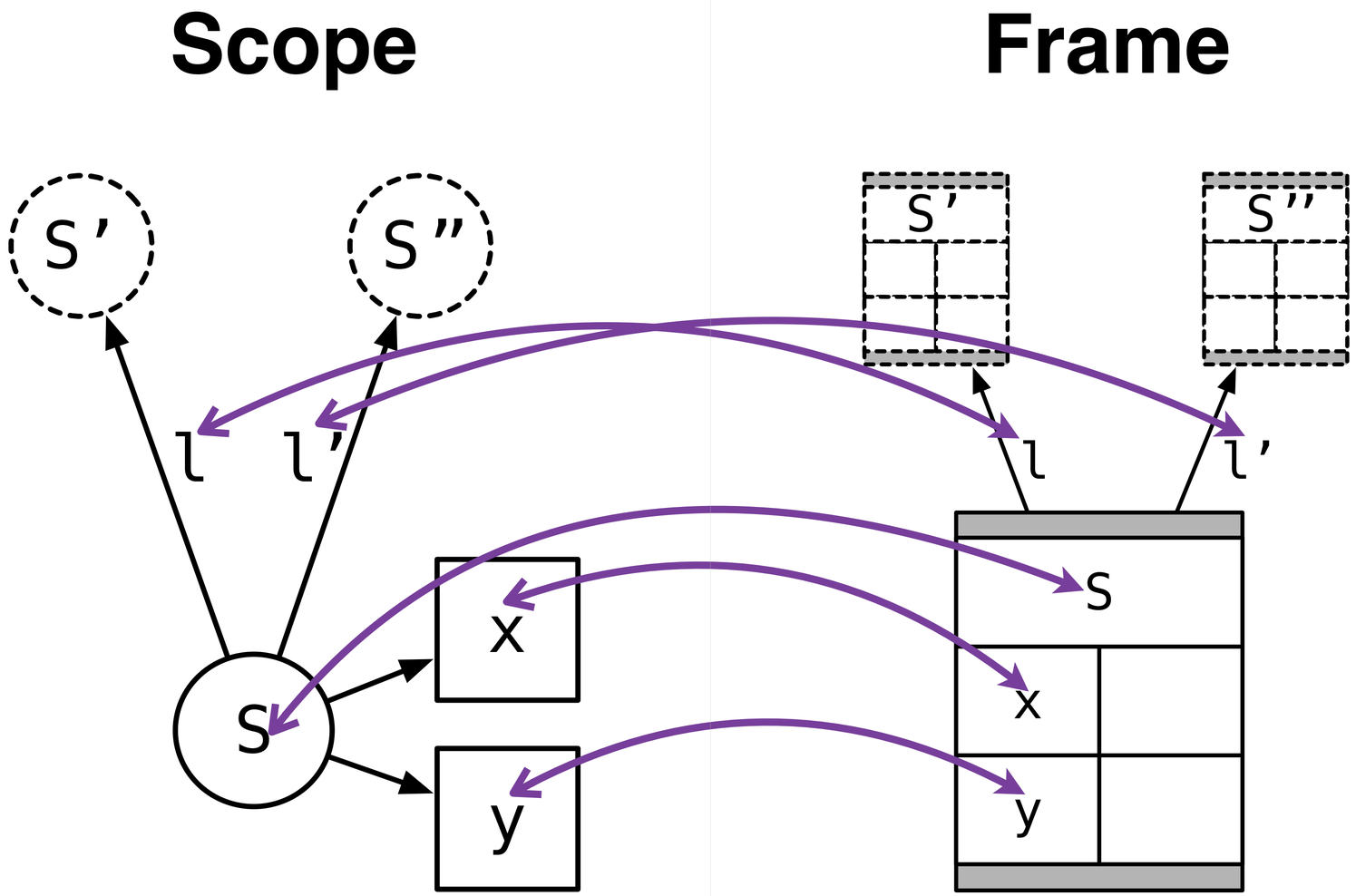
		Static	Dynamic
Lexical	`val x = 31;` `val y = x + 11;`		Substitution Environments De Bruijn Indices HOAS
Mutable	`var x = 31;` `x = x + 11;`		Stores/Heaps

Objects	`class A {`
	`  var x = 0;`
	`  var y = 42;`
	`}`
	`var r = new A();`
			Mutable Objects
			Stores/Heaps

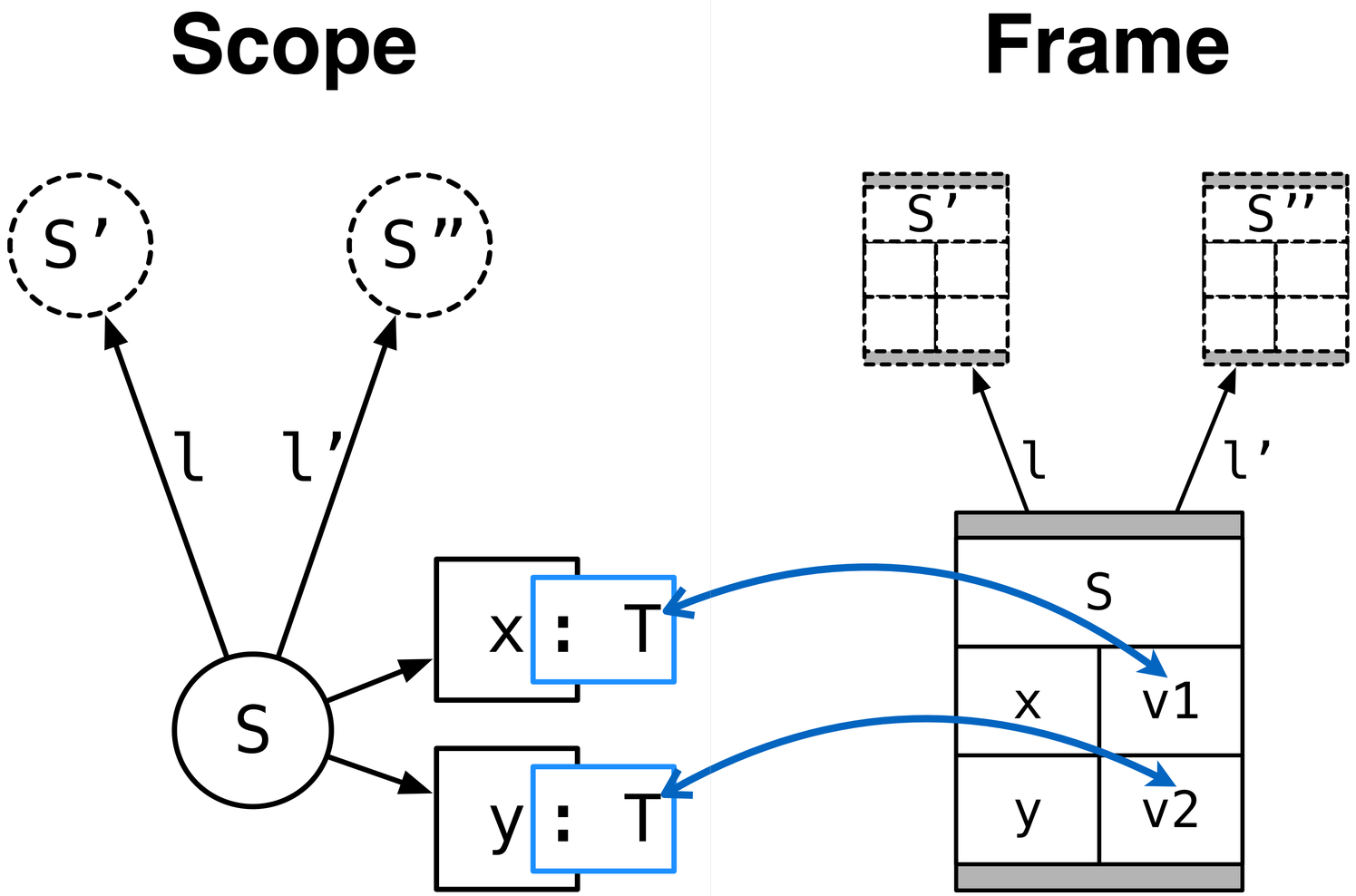
		Static	Dynamic
Lexical	`val x = 31;` `val y = x + 11;`		
Mutable	`var x = 31;` `x = x + 11;`		

Objects	`class A {`
	`  var x = 0;`
	`  var y = 42;`
	`}`
	`var r = new A();`
			

Well-Bound Frame

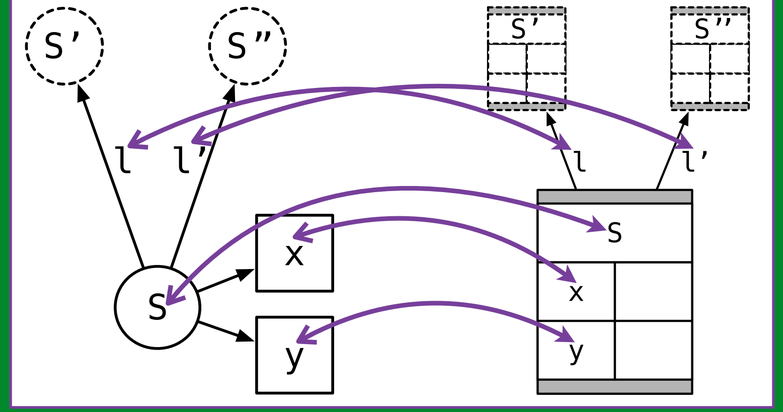


Well-Typed Frame

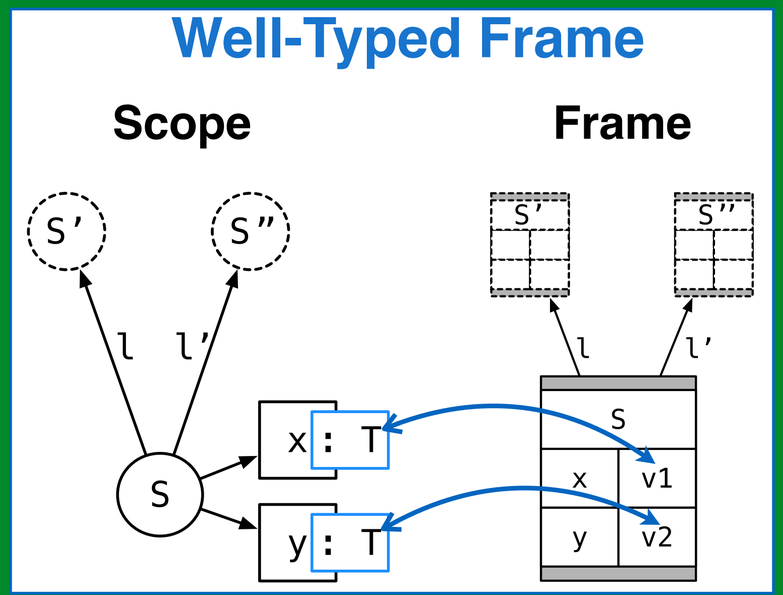


Good Frame Invariant

Well-Bound Frame

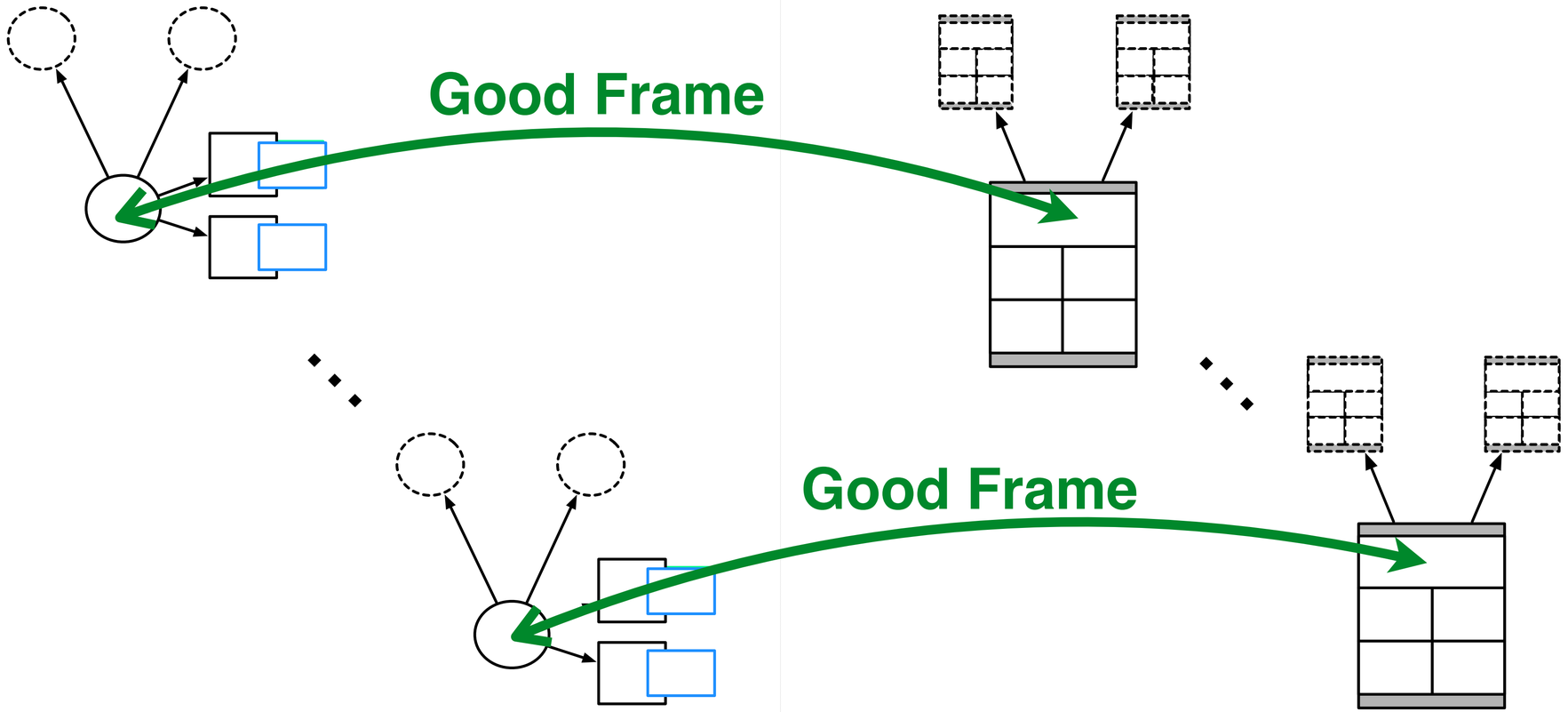


Well-Typed Frame

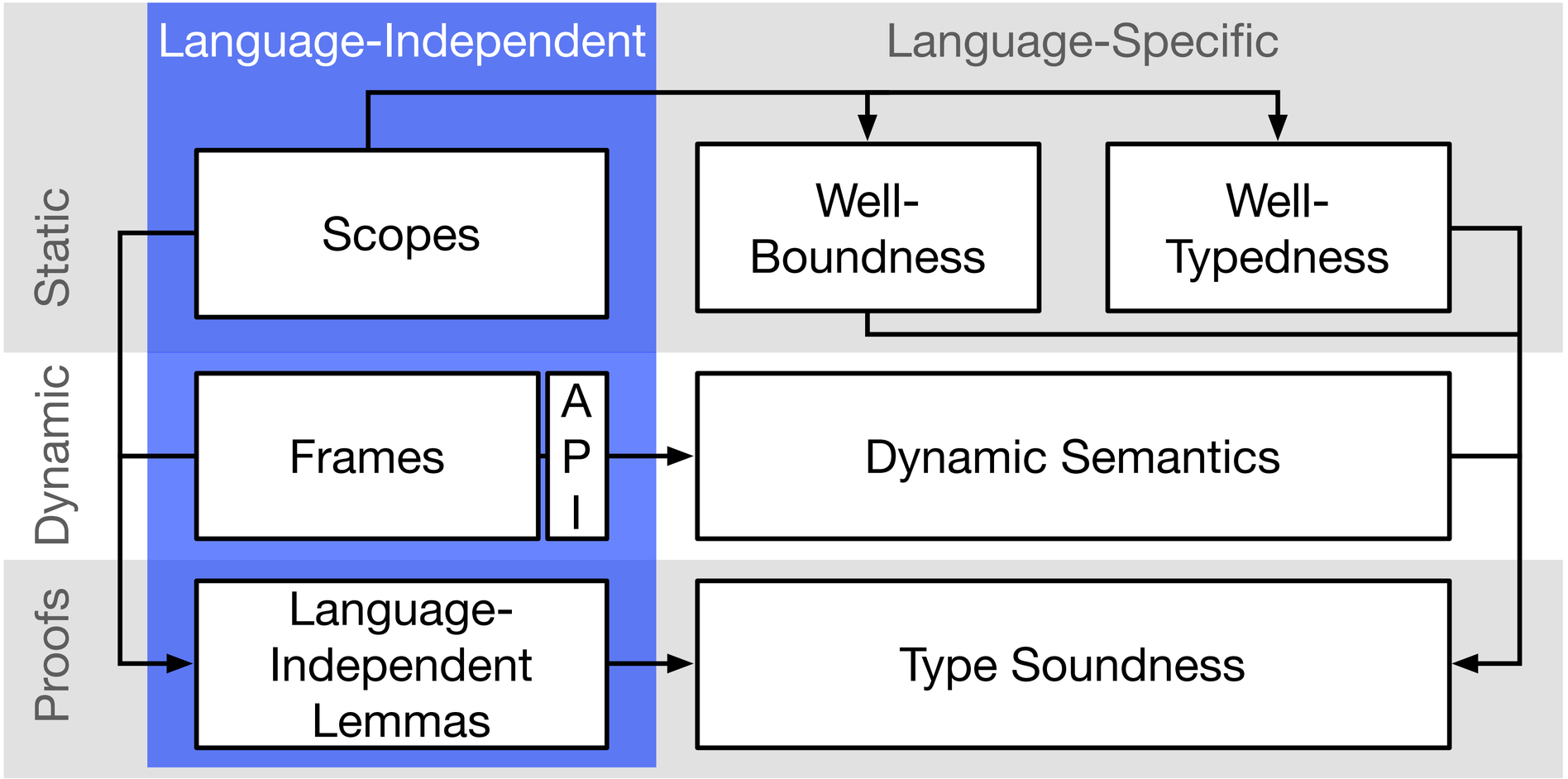


Good Heap Invariant

Every Frame is Well-Bound and Well-Typed

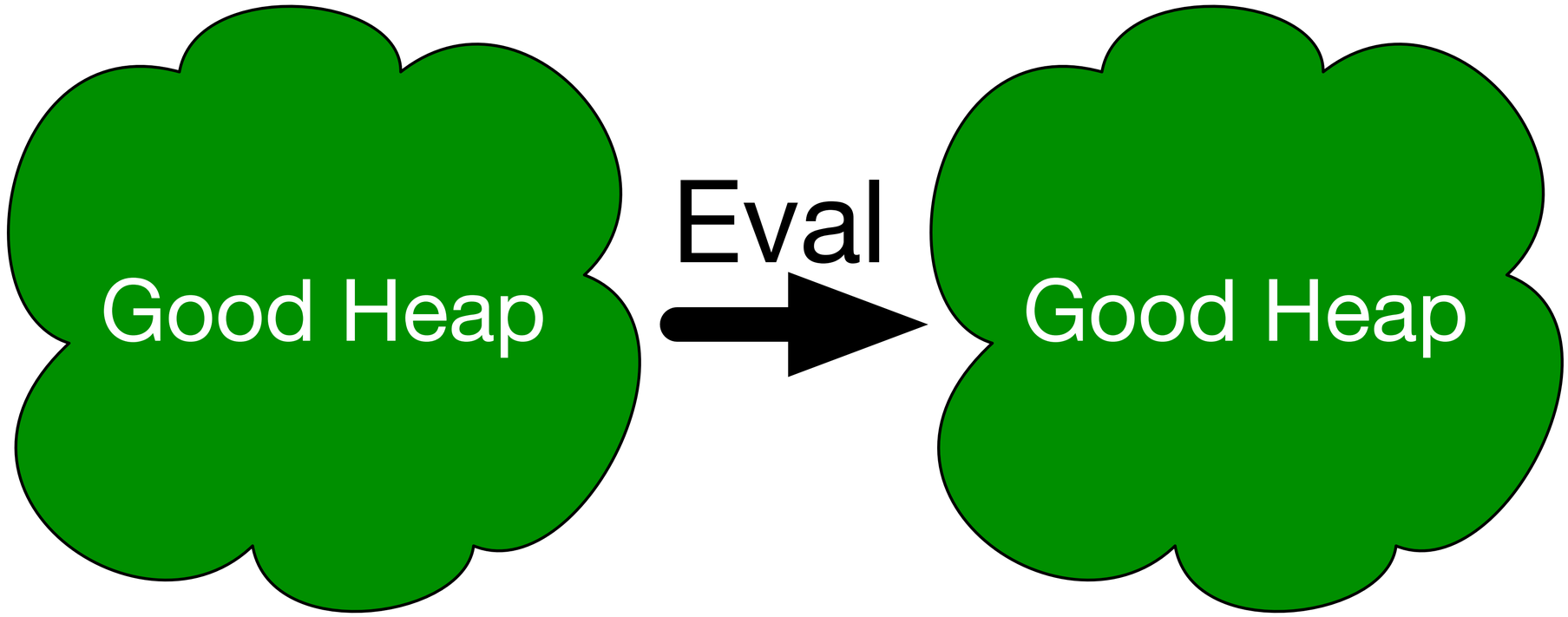


Architecture of a Specification



Type Soundness Principle

Evaluation Preserves
Good Heap Invariant



Some Research Projects

Specializing frame-based interpreters

- Encouraging results with interpreter for Tiger
- Does this scale to other languages / language features?

A frame-based virtual machine

- Common run-time for different languages
- Generic definition of garbage collection

Interpreters for declarative languages

- Grammar interpreters instead of parser generators?
- Interpreters for constraint-based languages?

Intrinsically-Typed Definitional Interpreters

TU Delft

A desirable property for programming languages is type safety: well-typed programs don't go wrong.

Demonstrating type safety for language implementations requires a proof. Such a proof is hard (at least tedious) for language models, and rarely done for language implementations.

Can we automatically check type safety for language implementations?

This paper shows how to do that at least for definitional interpreters for non-trivial languages. (By using scopes and frames to represent bindings.)

POPL 2018

https://doi.org/10.1145/3158104

Intrinsically-Typed Definitional Interpreters for Imperative Languages

CASPER BACH POULSEN, Delft University of Technology, The Netherlands
ARJEN ROUVOET, Delft University of Technology, The Netherlands
ANDREW TOLMACH, Portland State University, USA
ROBBERT KREBBERS, Delft University of Technology, The Netherlands
EELCO VISSER, Delft University of Technology, The Netherlands

A definitional interpreter defines the semantics of an object language in terms of the (well-known) semantics of a host language, enabling understanding and validation of the semantics through execution. Combining a definitional interpreter with a separate type system requires a separate type safety proof. An alternative approach, at least for pure object languages, is to use a dependently-typed language to encode the object language type system in the definition of the abstract syntax. Using such intrinsically-typed abstract syntax definitions allows the host language type checker to verify automatically that the interpreter satisfies type safety. Does this approach scale to larger and more realistic object languages, and in particular to languages with mutable state and objects?

In this paper, we describe and demonstrate techniques and libraries in Agda that successfully scale up intrinsically-typed definitional interpreters to handle rich object languages with non-trivial binding structures and mutable state. While the resulting interpreters are certainly more complex than the simply-typed λ-calculus interpreter we start with, we claim that they still meet the goals of being concise, comprehensible, and executable, while guaranteeing type safety for more elaborate object languages. We make the following contributions: (1) A *dependent-passing style* technique for hiding the weakening of indexed values as they propagate through monadic code. (2) An Agda library for programming with *scope graphs* and *frames*, which provides a uniform approach to dealing with name binding in intrinsically-typed interpreters. (3) Case studies of intrinsically-typed definitional interpreters for the simply-typed λ-calculus with references (STLC+Ref) and for a large subset of Middleweight Java (MJ).

CCS Concepts: • **Theory of computation → Program verification**; *Type theory*; • **Software and its engineering → Formal language definitions**;

Additional Key Words and Phrases: definitional interpreters, dependent types, scope graphs, mechanized semantics, Agda, type safety, Java

ACM Reference Format:
Casper Bach Poulsen, Arjen Rouvoet, Andrew Tolmach, Robbert Krebbers, and Eelco Visser. 2018. Intrinsically-Typed Definitional Interpreters for Imperative Languages. *Proc. ACM Program. Lang.* 2, POPL, Article 16 (January 2018), 34 pages. https://doi.org/10.1145/3158104

Authors' addresses: Casper Bach Poulsen, Delft University of Technology, The Netherlands, c.b.poulsen@tudelft.nl; Arjen Rouvoet, Delft University of Technology, The Netherlands, a.j.rouvoet@tudelft.nl; Andrew Tolmach, Portland State University, Oregon, USA, tolmach@pdx.edu; Robbert Krebbers, Delft University of Technology, The Netherlands, r.j.krebbers@tudelft.nl; Eelco Visser, Delft University of Technology, The Netherlands, e.visser@tudelft.nl.

2475-1421/2018/1-ART16
https://doi.org/10.1145/3158104