Scopes Describe Frames

A Uniform Model for Memory Layout in Dynamic Semantics

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

[1]
[2]
[3]

Problem with Previous Approaches

Static Name Binding

Handled in a number of ways in semantic specifications:

- *Lexical scope:* type substitution, type environments
- *Stateful references:* reference types, store typings
- *Structured memory:* class tables

Lack of uniform model

```
val x = 31;

val y = x + 11;
```

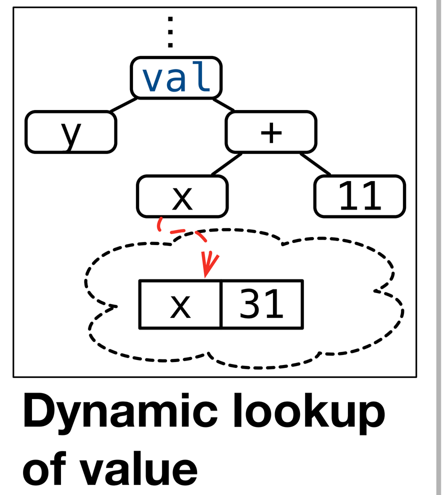
Statically resolved variable

Dynamic Memory

Handled in a number of ways in semantic specifications:

- *Lexical scope:* substitution, environments, de Bruijn, HOAS
- *Stateful references:* Mutable stores, heaps
- *Structured memory:* Mutable values for records, objects

Lack of uniform model



Dynamic lookup of value

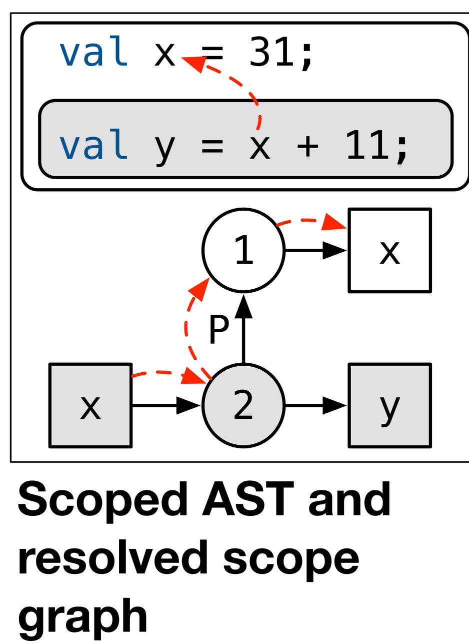
Our Solution

Scope Graphs [ESOP'15]

Nodes of scope graphs represent three basic notions derived from the program abstract syntax tree:

- *Scopes* (◯) and *edges* (◯→◯) between scopes
- *Declarations* (→□)
- *References* (□→)
- *Static resolution paths* (⇢)

Uniform model



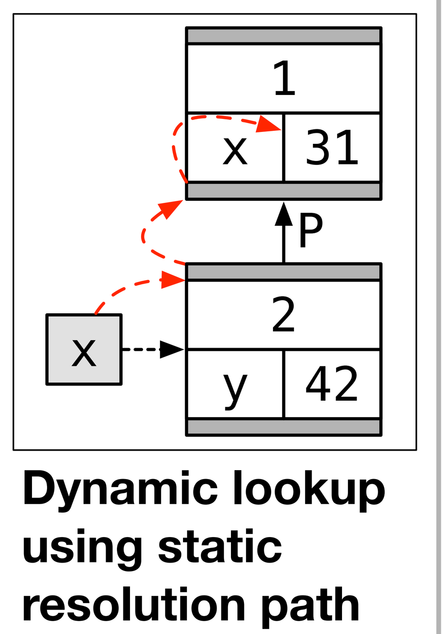
Scoped AST and resolved scope graph

Frames and Heaps

We propose frames as a language-independent model for dynamic memory. The model is based on these notions:

- *Frames* (▭) and *links* (▭→▭) between frames
- *Heap:* a frame graph
- *Dynamic lookup* (⇢): static resolution path interpreted relative to the "current" frame

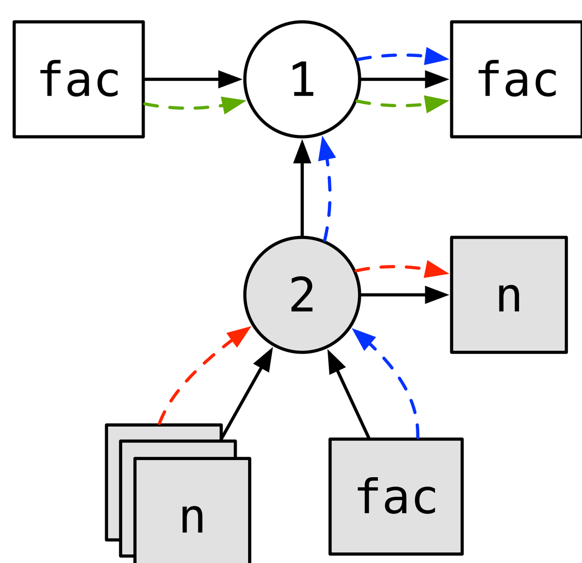
Uniform model

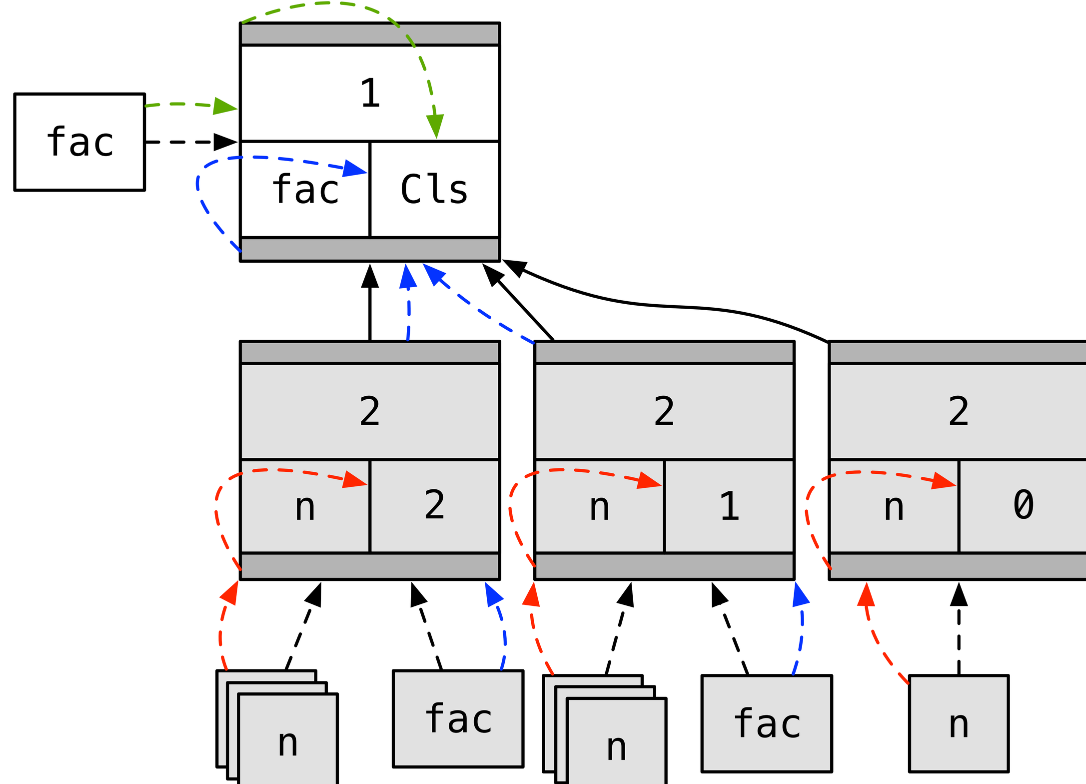


Dynamic lookup using static resolution path

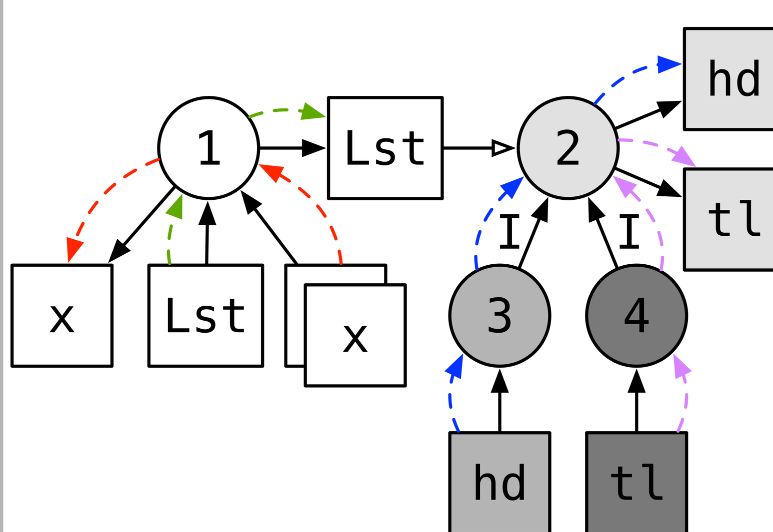
Static Binding matches Dynamic Behavior

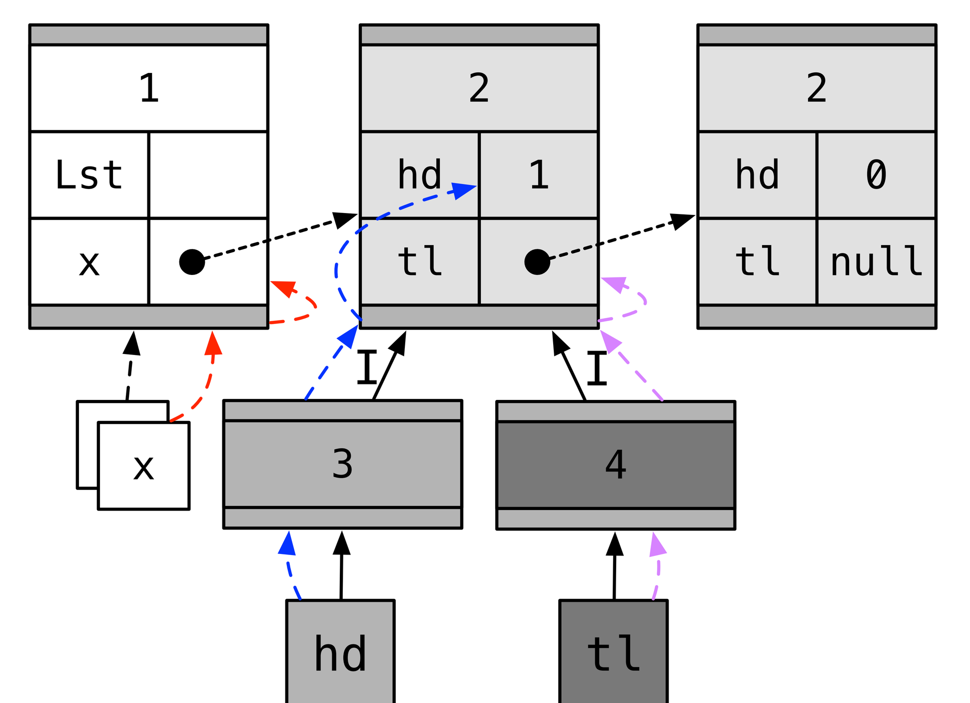
```
letrec fac =
  fun (n : Int) : Int {
    if (n == 0) {
      1
    } else {
      n * fac(n - 1)
    } }
in fac(2)
```






```
class Lst {
  hd : Int;
  tl : Lst;
}
var x = new Lst();
x.hd = 1;
x.tl = new Lst();
```

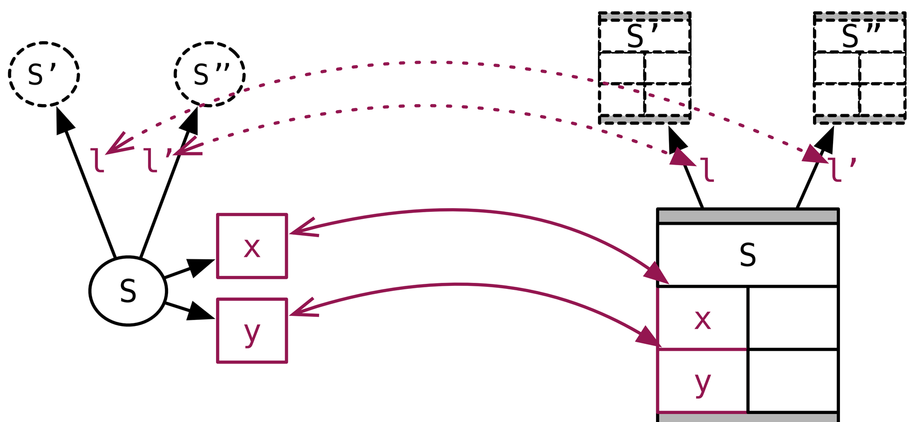




Memory Invariants

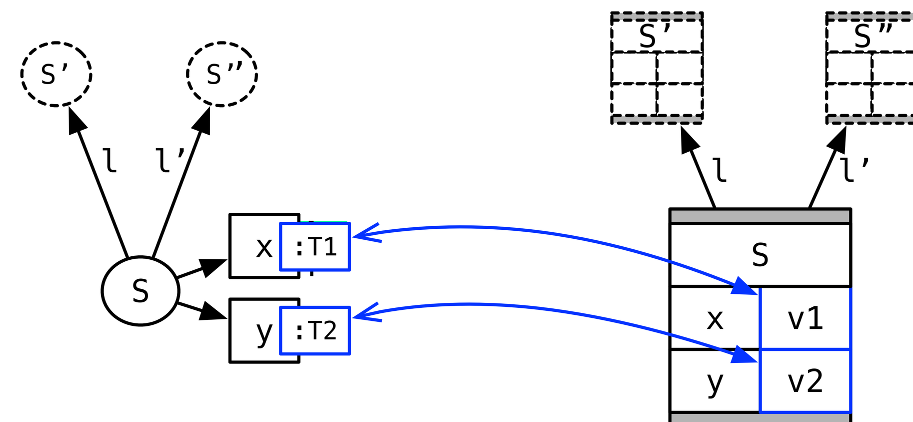
Well-Bound Frame

Frame slots and links correspond to scope declarations and edges



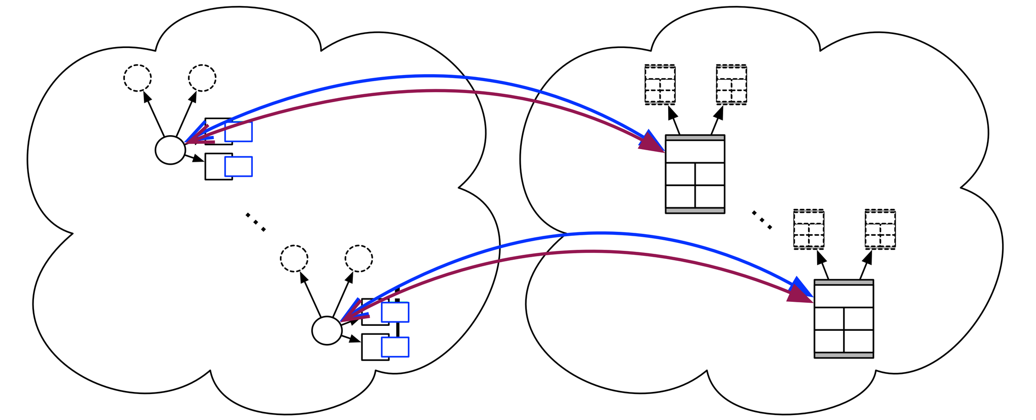
Well-Typed Frame

Types of values in frame slots match types of corresponding scope declarations



Good Heap

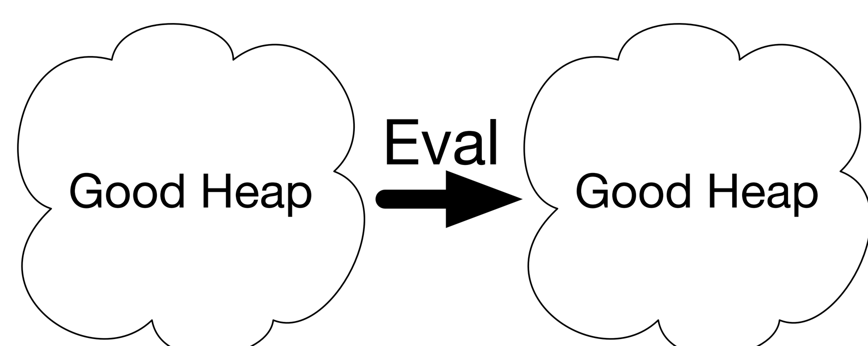
All frames are well-bound and well-typed



Verification

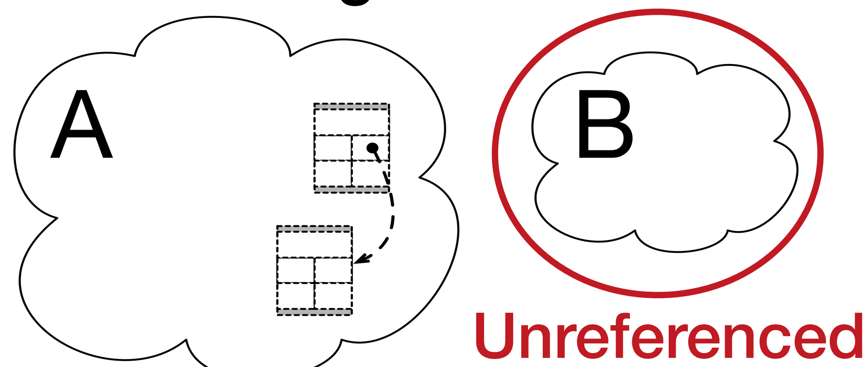
Type Soundness Principle

Evaluation preserves good heap property



Theorem (Type Soundness).
For a well-bound and well-typed program in a good heap, evaluation gives a well-typed value and good heap

Garbage Collection



Lemma (Safe Removal).
For a good heap X = A ∪ B, if nothing in B is referenced from A, then A is a good heap (B can be safely garbage collected).

Specification Architecture