Frame VM

A virtual machine using scopes as frames

Chiel Bruin

May 15, 2019

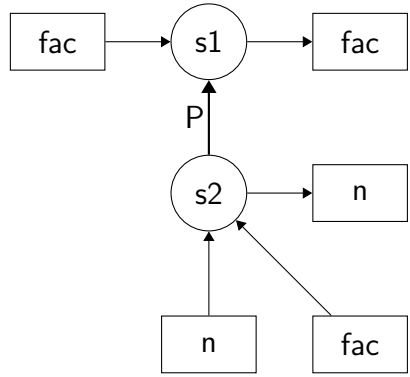
TU Delft

Outline

- Scopes as frames
  - Scope graphs
  - Memory layout
- Frame VM design
  - Instructions
- Continuation Passing Style
  - VM sugar
- Demo / Examples
- Call/cc
  - Scheme
- What next

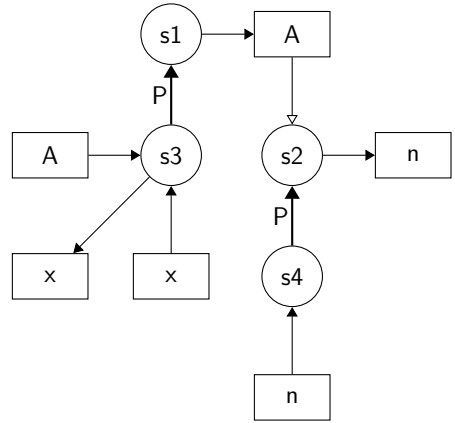
Scope graphs - functions

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

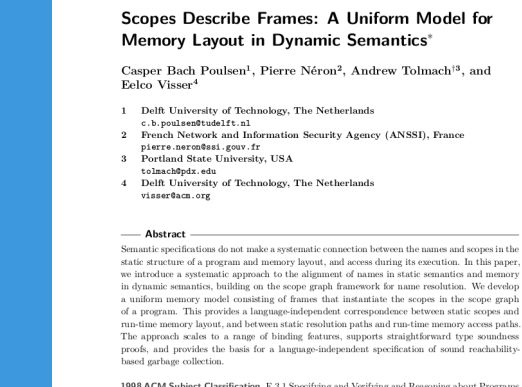


Scope graphs - records

```
record A {
    n : Int
}

begin
    var x : A;
    x = new A
    x.n = 1
```



Scopes describe frames



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

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

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

Abstract

Semantic specifications do not make a systematic connection between the names and scopes in the static structure of a program and memory layout, and access during its execution. In this paper, we introduce a systematic approach to the alignment of names in static semantics and memory in dynamic semantics, building on the scope graph framework for name resolution. 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.

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

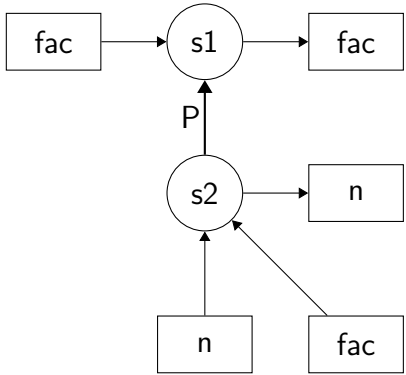
Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.20

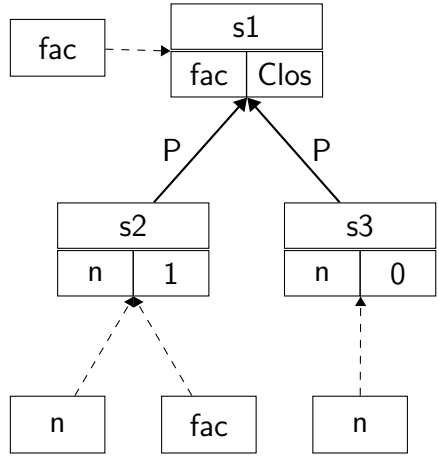
Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.2.1.10

1 Introduction

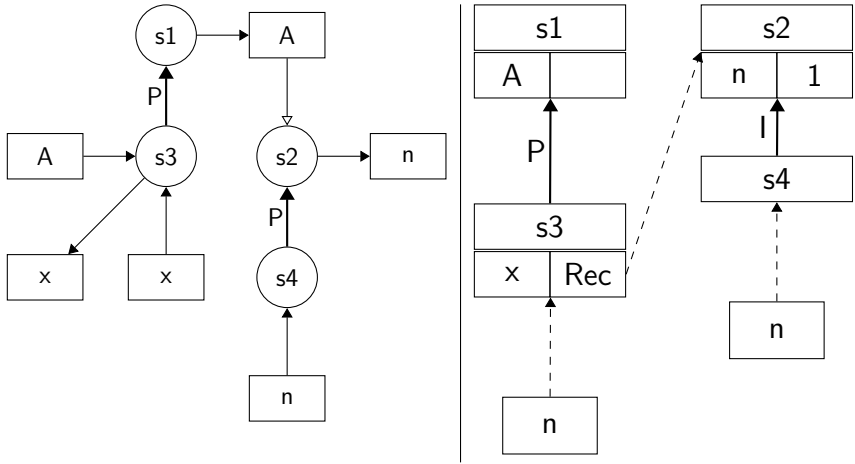
- Scope graph ⇔ memory layout
- Resolution paths ⇔ memory access

Scopes as frames - functions





Scopes as frames - records

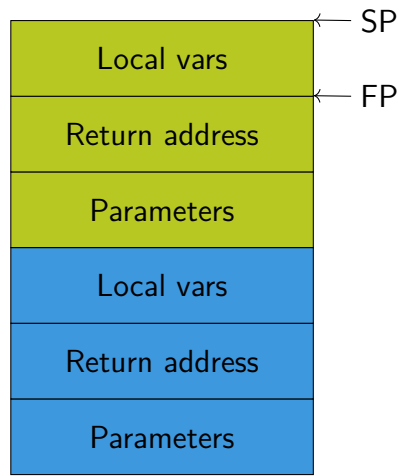


Scopes as frames

- Language independent representation of memory layout
  - Data sharing between languages
  - Garbage collection
- Simplify code generation by using scope graph
  - Given a Dynsem spec and scope graph, get a compiler
  - Control flow

Conventional machine

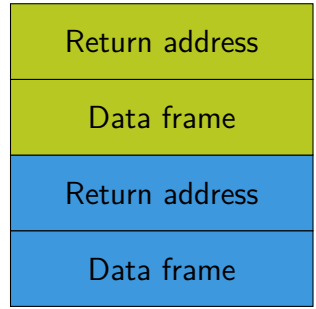
- Languages like C use a heap and (call)stack
- The stack stores local variables, parameters and return addresses
- The heap stores data frames
- Elements on the heap are accessed using pointers stored on the stack



Frames as memory

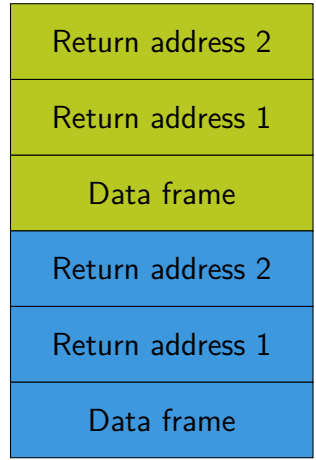
- Local variables are in the scope graph
- Parameters are in the scope graph

$\Rightarrow$ Replaced by a (data) frame



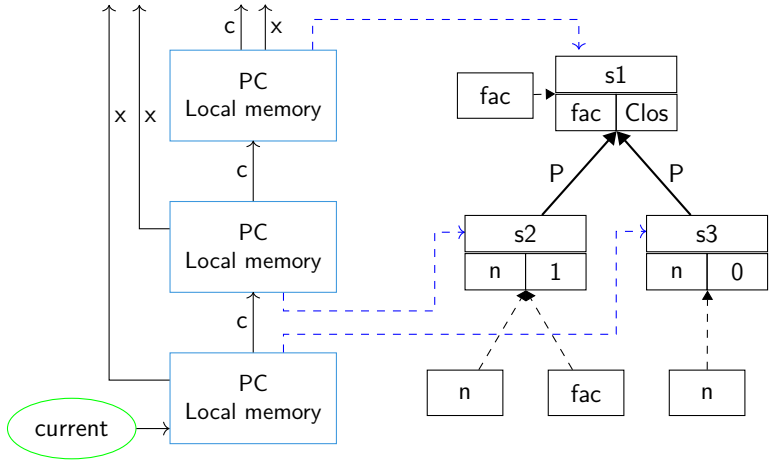
Control frames

- Why only have one return address?
- Having multiple allows for:
  - Exceptions
  - Search algorithms
  - Exit



VM overview

Combining the control graph and data frames into a single system



VM modes

- Two modes
- Stacy
  - Stack machine
  - Used in this presentation
- Roger
  - Register machine
  - Higher level, easier to compile to
  - Early in development

Blocks

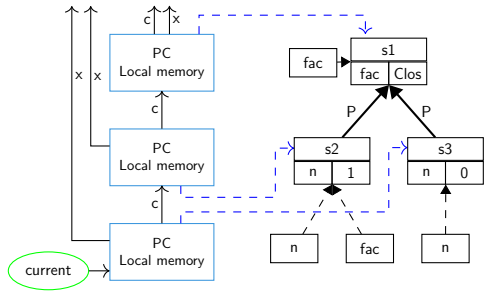
- A program consists of blocks
- A block has a label
- A block ends on jumping instruction

```
MAIN:
  ipush 1
  ipush 2
  addi
  print
  jump LBL
```

Instructions

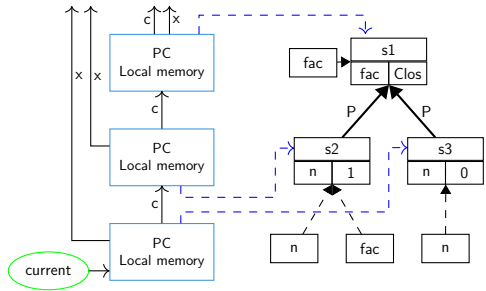
- Arithmetic, stack, ...

```
ipush 1
addi
dup
swap

print
debug
```



Instructions

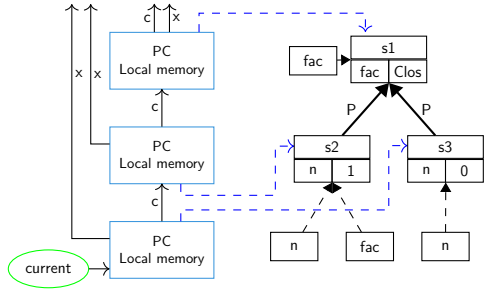
- Current data frame
  `mkcurrent`
  `getcurrent`
  `get [P, 0]`
  `set [0]`
  `link [] P`
- Data frame
  `getr [P, 0]`
  `setr [0]`
  `linkr P`
  `new`



Instructions

- Control frame

```
jump LBL
jumpz THEN ELSE
cget [c]
cset [c]
ccall LBL
transfer 1
```



Continuation Passing Style

- Continuations can be seen as *the rest of the computation*
- In CPS this next step is made explicit

```
(+ 1 (+ 2 3))
```

```
(+' 2 3 (lambda (x) (+ 1 x)))
```

CPS function calls

```
(define inc
    (lambda (x)
        (+ 1 x)
    )
)
```

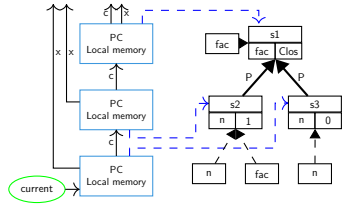
```
(+ 1 (inc 40))
```

```
(define inc'
    (lambda (x, k)
        (k (+ 1 x))
    )
)
```

```
(inc' 40 (lambda (x)
    (+ 1 x)
))
```

CPS and the Frame VM

- Continuation of execution $\rightarrow$
  Some scope, PC and local memory
- Return addresses are continuations*
- Bookkeeping is required



CPS exceptions

```
(define func (lambda (n k x)
  (if (> n 0)
    (k (+ n 1))
    (x 'error)
  )
))

(func 41 (lambda (x) x) (lambda (x) x))
```

Sugar

- Takes care of the bookkeeping
- Designed for a language with exceptions

```
newscope P
exitscope [P]
call FUNC RET
return
try TRY CATCH LBL
throw
```

Sugar

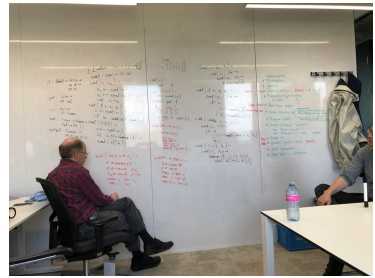
```
call FUNC RET ⇒

cnew FUNC 2
dup
cget [x]
csetr [x]
dup
cget []
csetr [c]
ccall RET
```

Examples

- Records
- Functions
- Recursion
- Tail-recursion
- Debugging

Code generation

- Handcrafted bytecode
  - Exception handling
  - Generator functions
  - Cooperative multithreading
- Scheme compiler*
- LEP projects
  - Prolog
  - Rust
  - Haskell
- Dynsem
- Dynamix



TU Delft

Call/cc

- Call-with-current-continuation
- Get the current execution state
  (The explicit rest of the computation in CPS)
- This state can be called like a function
- Making a sandwich
- Expensive operation

Call/cc in Scheme

```
(define add-two #f)
(+ 2 (call-cc (
  lambda (cont)
    (set! add-two cont)
    3
  ))
)
(add-two 10)
(add-two 40)
```

What is the next instruction?

Call/cc in Scheme

```
(define add-x #f)
(define x 6)
(+ x (call-cc (
  lambda (cont)
    (set! add-x cont)
    3
  ))
)
(add-x 10)
(set! x 2)
(add-x 40)
```

What should this do?

Call/cc in Scheme

```
(define add-x #f)
(define x 6)
(+ (call-cc (
  lambda (cont)
    (set! add-x cont)
    3
  )) x
)
(add-x 10)
(set! x 2)
(add-x 40)
```

And what about this?

Call/cc issues

- The big question is what data needs to be copied
  - Elements on the stack
  - Continuation slots
  - Current data frame
  - Children of the data frame
  - Parents of the data frame
  - Global variables
- These policies are not yet implemented

What next?

- Modeling memory copy operations
- Modules
- Register machine
- Dynamix
- Stratego interpreter to actual VM

TU Delft