From Neil’s Shoulders to the Moon

Lua

Jan Midtgaard
Aarhus University
State of affairs, PL-wise

Good news:

Today’s popular programming languages (JavaScript, Python, Lua, ...) are both
□ dynamically typed and
□ higher order

Bad news:

In terms of software guarantees, that means we are not sure
□ what kind of arguments functions can expect, and
□ where control may transfer to
Neil to the rescue!

Neil pioneered static analysis techniques to solve these problems well before the rest of us. In particular:

- **Token-based abstraction** (POPL’79, MJ’81, POPL’82, with Muchnick) to statically approximate the origin of heap-allocated data, and

- **Control-flow analysis** (ICALP’81) to statically approximate the origin of higher-order functions

Programs as data objects indeed :-)
Neil, the control-flow analysis (CFA) pioneer

CFA alone has been the subject of much research — see CFA survey (Midtgaard:CompSurveys12)

200+ articles:

(just to name a few)

Many people with roots in TOPPS, DIKU and DK!
It’s for real!

There is continuous static type analysis work in

- Just-in-Time compilers (Google’s V8, LuaJIT, . . .) to improve code generation and speed up execution,

- Development environments (Eclipse, IntelliJ, . . .) to provide programmer feedback.
The rest of this talk

A concrete example of Neil’s contributions put to use:

Developing a static type analysis
for the Lua programming language
Lua, in brief

- Developed in Brazil, hence the name (‘lua’ = ’moon’)
- Lightweight language (few, well-chosen features):
  - Dynamically typed
  - First-class functions
  - Builtin tables (associative arrays)
  - ...
- Multiparadigm (FP, OO, ...)
- Lightweight, cross-platform implementation
- Standalone and easily embeddable
- ...
Example: Sets of numbers (Ierusalimschy’13)

Set = {}  

-- create a new set with the values of a given list
function Set.new (l)
    local set = {}
    for __, v in ipairs(l) do set[v] = true end
    return set
end

function Set.union (a, b)
    local res = Set.new{}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

s1 = Set.new{10,20,30,50}
s2 = Set.new{30,1}
s3 = Set.union(s1,s2)
Example: Sets of numbers, desugared

Set = {}

-- create a new set with the values of a given list
Set.new = function (l)
    local set = {}
    for _, v in ipairs(l) do set[v] = true end
    return set
end

Set.union = function (a, b)
    local res = Set.new{}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

s1 = Set.new{10,20,30,50}
s2 = Set.new{30,1}
s3 = Set.union(s1,s2)
Example: Sets of numbers, desugared

Set = {}

-- create a new set with the values of a given list
Set.new = function (l)
    local set = {}
    for _, v in ipairs(l) do set[v] = true end
    return set
end

Set.union = function (a, b)
    local res = Set.new{}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
s3 = Set.union(s1, s2)
Example: Sets of numbers, desugared

Set = {}  

-- create a new set with the values of a given list
Set.new = function (l)
    local set = {}
    for _, v in ipairs(l) do set[v] = true end
    return set
end

Set.union = function (a, b)
    local res = Set.new{}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
s3 = Set.union(s1, s2)
Example: Sets of numbers, desugared

Set = {}  

-- create a new set with the values of a given list
Set.new = function (l)
    local set = {}
    for _, v in ipairs(l) do set[v] = true end
    return set
end

Set.union = function (a, b)
    local res = Set.new{}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
s3 = Set.union(s1, s2)
Example: Sets of numbers, desugared

Set = {}

-- create a new set with the values of a given list
Set.new = function (l)
    local set = {}
    for _, v in ipairs(l) do set[v] = true end
    return set
end

Set.union = function (a, b)
    local res = Set.new{}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
s3 = Set.union(s1, s2)
Set = {} -- create a new set with the values of a given list
Set.new = function (l)
    local set = {}
    for _, v in ipairs(l) do set[v] = true end
    return set
end

Set.union = function (a, b)
    local res = Set.new{}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
s3 = Set.union(s1, s2)
Lua in a bit more detail... 

To add PL spice it also includes:

- **Metatables with events and metamethods:**
  These are used to model both OO-like inheritance and overriding

- Proper tail calls

- Co-routines

- ...
Example: Sets w/overriding (Ierusalimschy’13)

Set = {}

local mt = {} -- metatable for sets

-- create a new set with the values of a given list
function Set.new (l)
  local set = {}
  setmetatable(set, mt)
  for _, v in ipairs(l) do set[v] = true end
  return set
end

function Set.union (a, b)
  local res = Set.new{}
  for k in pairs(a) do res[k] = true end
  for k in pairs(b) do res[k] = true end
  return res
end

mt.__add = Set.union

s1 = Set.new{10,20,30,50}
s2 = Set.new{30,1}
s3 = s1 + s2
Example: Sets w/overriding (Ierusalimschy’13)

Set = {}

local mt = {} -- metatable for sets
declare a metatable

-- create a new set with the values of a given list
function Set.new (l)
    local set = {}
    setmetatable(set, mt)
    for __, v in ipairs(l) do set[v] = true end
    return set
end

function Set.union (a, b)
    local res = Set.new{}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

mt.__add = Set.union

s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
s3 = s1 + s2
Example: Sets w/overriding (Ierusalimschy’13)

Set = {}

local mt = {} -- metatable for sets

-- create a new set with the values of a given list
function Set.new (l)
    local set = {}
    setmetatable(set, mt)
    for _, v in ipairs(l) do set[v] = true end
    return set
end

function Set.union (a, b)
    local res = Set.new{}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

mt.__add = Set.union

s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
s3 = s1 + s2
Example: Sets w/overriding (Jerusalimschy’13)

Set = {}

```lua
local mt = {} -- metatable for sets

-- create a new set with the values of a given list
function Set.new (l)
    local set = {}
    setmetatable(set, mt)
    for _, v in ipairs(l) do set[v] = true end
    return set
end

function Set.union (a, b)
    local res = Set.new{}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

mt.__add = Set.union
```

s1 = Set.new{10,20,30,50}
s2 = Set.new{30,1}
s3 = s1 + s2
Example: Sets w/overriding (Ierusalimschy’13)

Set = {}

local mt = {} -- metatable for sets

-- create a new set with the values of a given list
function Set.new (l)
    local set = {}
    setmetatable(set, mt)
    for _, v in ipairs(l) do set[v] = true end
    return set
end

function Set.union (a, b)
    local res = Set.new{}
    for k in pairs(a) do res[k] = true end
    for k in pairs(b) do res[k] = true end
    return res
end

mt.__add = Set.union

s1 = Set.new{10, 20, 30, 50}
s2 = Set.new{30, 1}
s3 = s1 + s2
Using some of Neil’s terminology (MJ’81), we build a

- forward,
- flow-sensitive,
- attribute-independent,
- monomorphic

flow analysis over a suitable lattice structure.
A lattice for Lua type analysis

Akin to TAJJS lattice for JavaScript (Jensen-al:SAS09)
A lattice for Lua type analysis

Akin to TAJJS lattice for JavaScript (Jensen-al:SAS09)
A lattice for Lua type analysis

Akin to TAJS lattice for JavaScript (Jensen-al:SAS09)
A lattice for Lua type analysis

Akin to TAJS lattice for JavaScript (Jensen-al:SAS09)
A lattice for Lua type analysis

Akin to TAJS lattice for JavaScript (Jensen-al:SAS09)
A lattice for Lua type analysis

Akin to TAJS lattice for JavaScript (Jensen-al:SAS09)
A lattice for Lua type analysis

Akin to TAJS lattice for JavaScript (Jensen-al:SAS09)
Transfer functions over this lattice now models Lua’s semantics.

The manual specifies, e.g., events with interpreters:

```lua
function add_event (op1, op2)
    local o1, o2 = tonumber(op1), tonumber(op2)
    if o1 and o2 then  -- both operands are numeric?
        return o1 + o2  -- ‘+’ here is the primitive ‘add’
    else  -- at least one of the operands is not numeric
        local h = getbinhandler(op1, op2, "__add")
        if h then
            -- call the handler with both operands
            return (h(op1, op2))
        else  -- no handler available: default behavior
            error(···)
        end
    end
end
```
Transfer functions over this lattice now models Lua’s semantics.

The manual specifies, e.g., events with interpreters:

```lua
function add_event (op1, op2)
    local o1, o2 = tonumber(op1), tonumber(op2)
    if o1 and o2 then  -- both operands are numeric?
        return o1 + o2  -- '+' here is the primitive 'add'
    else              -- at least one of the operands is not numeric
        local h = getbinhandler(op1, op2, "__add")
        if h then
            -- call the handler with both operands
            return (h(op1, op2))
        else          -- no handler available: default behavior
            error("...")
        end
    end
end
```
Transfer functions over this lattice now models Lua’s semantics.

The manual specifies, e.g., events with interpreters:

```lua
function add_event (op1, op2)
    local o1, o2 = tonumber(op1), tonumber(op2)
    if o1 and o2 then -- both operands are numeric?
        return o1 + o2  -- ’+’ here is the primitive ’add’
    else -- at least one of the operands is not numeric
        local h = getbinhandler(op1, op2, "__add")
        if h then
            -- call the handler with both operands
            return h(op1, op2)
        else -- no handler available: default behavior
            error("...")
        end
    end
end
```
Transfer functions over this lattice now models Lua’s semantics.

We try to stick as close to the manual as possible:

```plaintext
and transfer_arith_event clab info op event op1 op2 =
let o1, o2 = VL.coerce_tonum op1, VL.coerce_tonum op2 in
mvl_join
  (if VL.may_be_number (VL.meet o1 o2) (* -- both operands are numeric? *)
   then red_return (VL.binop op o1 o2) (* -- ‘+’ here is the primitive ‘add’ *)
   else merror)
  (getbinhandler op1 op2 event >>= fun h ->
   if VL.may_be_proc h
   then (* -- call the handler with both operands *)
     transfer_calls clab h [op1;op2] info >>= adjust_to_single
   else (* -- no handler available: default behavior *)
     merror)
```
Transfer functions over this lattice now models Lua’s semantics.

We try to stick as close to the manual as possible:

```plaintext
and transfer_arith_event clab info op event op1 op2 =
  let o1, o2 = VL.coerce_tonum op1, VL.coerce_tonum op2 in
  mvl_join
    (if VL.may_be_number (VL.meet o1 o2) (* -- both operands are numeric? *)
    then red_return (VL.binop op o1 o2) (* -- '+' here is the primitive 'add' *)
    else merror)
  (getbinhandler op1 op2 event >>= fun h ->
   if VL.may_be_proc h
   then (* -- call the handler with both operands *)
     transfer_calls clab h [op1;op2] info >>= adjust_to_single
   else (* -- no handler available: default behavior *)
     merror)
```

numeric add
Transfer functions over this lattice now models Lua’s semantics.

We try to stick as close to the manual as possible:

```plaintext
and transfer_arith_event clab info op event op1 op2 =
  let o1, o2 = VL.coerce_tonum op1, VL.coerce_tonum op2 in
  mvl_join
    (if VL.may_be_number (VL.meet o1 o2) (* -- both operands are numeric? *)
      then red_return (VL.binop op o1 o2) (* -- ‘+’ here is the primitive ‘add’ *)
      else merror)
    (getbinhandler op1 op2 event >>= fun h ->
      if VL.may_be_proc h
        then (* -- call the handler with both operands *)
          transfer_calls clab h [op1;op2] info >>= adjust_to_single
        else (* -- no handler available: default behavior *)
          merror)
```

numeric add

overridden add
Transfer functions over this lattice now models Lua’s semantics.

We try to stick as close to the manual as possible:

```ocaml
and transfer_arith_event clab info op event op1 op2 =
  let o1, o2 = VL.coerce_tonum op1, VL.coerce_tonum op2 in
  mvl_join
    (if VL.may_be_number (VL.meet o1 o2) (* -- both operands are numeric? *)
     then red_return (VL.binop op o1 o2) (* -- '+' here is the primitive 'add' *)
     else merror)
  (getbinhandler op1 op2 event >>= fun h ->
   if VL.may_be_proc h
     then (* -- call the handler with both operands *)
       transfer_calls clab h [op1; op2] info >>= adjust_to_single
     else (* -- no handler available: default behavior *)
       merror)
```

Prototype in approx. 5000 lines of OCaml
Implemented as AST walker, written monadically
[Demo]
Future work

For now, focus has been on correctness and completeness.

However, I have grand plans for

- eliminating interpretive overhead ("poor man’s partial evaluation")
- localization experiments
- experiments with data structures (Patricia trees, . . .)
- forwards/backwards analysis for abstract debugging (Deutsch:PLDI93)

(I’m planning to release the software shortly)
Another prominent user of Lua is Wikipedia.

100+ Lua modules/scripts for
- language,
- encodings,
- HTML-rendering,
- strings,
- JSON,
- ...
Speaking of Wikipedia... 

The online encyclopedia has entries for several prominent Danish computer scientists:

- Dines Bjørner
- Peter Naur
- Mikkel Thorup
- Mads Tofte
- ...

But it is clearly missing an entry.
Here it is

(It’s a work in progress)

In the Wikipedia spirit, I encourage you adjust and extend it as you see fit*

*Thank you Amir for already having done so!
Summary and conclusion

The static type analysis is a concrete example of (a fraction of) Neil’s work put to use.
Summary and conclusion

The static type analysis is a concrete example of (a fraction of) Neil’s work put to use.

Neil:

☐ Standing on your shoulders, I can reach for the moon... :-}
The static type analysis is a concrete example of (a fraction of) Neil’s work put to use.

Neil:

☐ Standing on your shoulders, I can reach for the moon... :-)  

☐ Your tireless effort and your career is exemplar and an inspiration to younger scholars.
Summary and conclusion

The static type analysis is a concrete example of (a fraction of) Neil’s work put to use.

Neil:

☐ Standing on your shoulders, I can reach for the moon... :-)

☐ Your tireless effort and your career is exemplar and an inspiration to younger scholars.

☐ Thank you, for your lasting, time less contributions to static analysis (and programming languages).
Summary and conclusion

The static type analysis is a concrete example of (a fraction of) Neil’s work put to use.

Neil:

☐ Standing on your shoulders, I can reach for the moon... :-)

☐ Your tireless effort and your career is exemplar and an inspiration to younger scholars.

☐ Thank you, for your lasting, time less contributions to static analysis (and programming languages).

Thanks