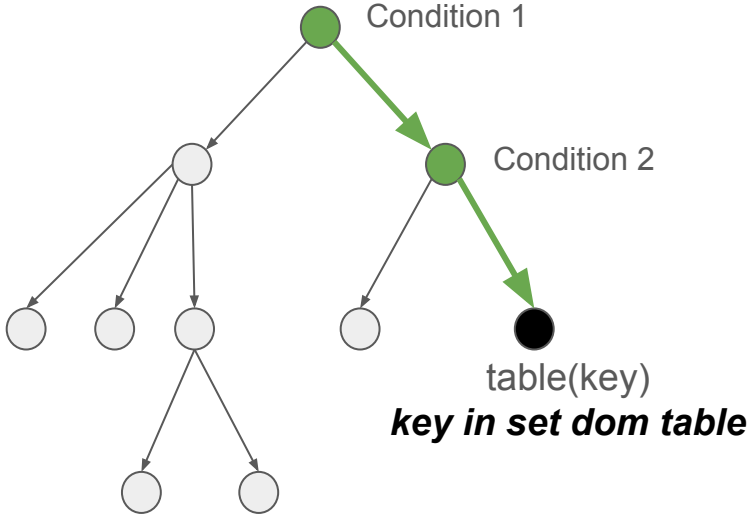
Towards Operation POG for VDM

Nick Battle and Peter Gorm Larsen

Proof Obligations (within functions)

```
func(key:K, table:T) ==
```




```
pre key... table...
```

- Functions are simple expression *trees*
- Unambiguous paths to POs
- General form of POs is…

```
for all possible valid args...
  context stack of path...
    conditions leading to...
      the base obligation
```

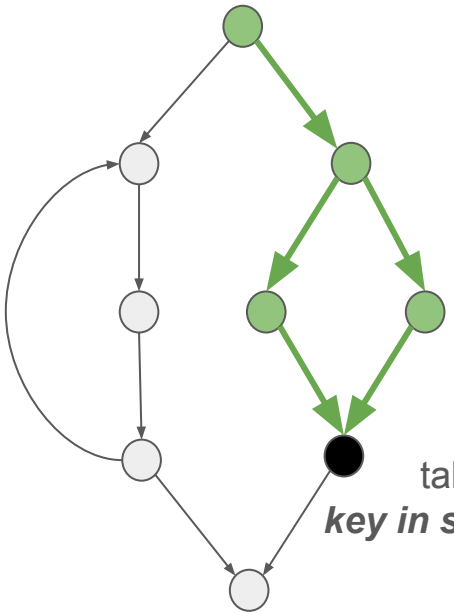
e.g.

```
forall key:K, table:T &
  pre_func(key, table) =>
    condition 1 =>
      condition 2 =>
        key in set dom table
```

Proof Obligations (within operations)

```
state Sigma of
    table : T
end

op(key:K) ==
```



table(key)

key in set dom table

- Operations have *control flow graphs*
- Can have *alternative* paths to POs
- Can have loops and exceptions
- Can update state, including "dcl" state
- Can call operations (inc. other modules)
- Can have multiple return points

```
for all possible args
  for all possible states
    for all possible paths
      context stack of conditions...
        and loops and exceptions...
          and state changes...
            and operation calls leading to...
              the base obligation
```

Proof Obligations (within operations)

- Current operation POG gives the *base obligation* in explicit operations
  - Like "***`key in set dom table`***" with no path context
  - Always "Unchecked" (ignored)
  - VDM *method* uses operation decomposition and data reification
- New operation POG is better, but still *incomplete*
  - Solved the easy cases, harder ones remain…
  - But VDM-SL "Unchecked" POs down from 21% to 9.6%

- Turns possible *paths* through the graph into PO context expressions
  - Paths reflect *possible histories* of updated variables
- Top level "forall" adds ***`mk_Sigma(v1, v2, ...):Sigma`*** for state
  - Means "*for any combination of arguments and module state…*"
- State "`:=`" assignments create a "let" context to update the value
  - New values *hide* the top level "forall" values

Proof Obligations (assignments)

- State updates add "let" contexts to the PO
- Update values *hide* earlier values

```
state Sigma of
    sv : nat
end

op(z:nat) r:real ==
  if z > 10 then
  (
      sv := z;
      sv := sv * 2;
      return 1/sv    -- PO#1
  )
  else
      ...
pre z > sv;
```

```
Proof Obligation 1: (Unproved)
(forall z:nat, mk_Sigma(sv):Sigma &
  pre_op(z, mk_Sigma(sv)) =>
    ((z > 10) =>
      (let sv : nat = z in
        (let sv : nat = (sv * 2) in
          sv <> 0))))
```

Proof Obligations (complex assignments)

- More complex assignments modify values in the "let"
- Sequence and map designators use "++", field designators use "mu"

```
state Sigma of
    sv : seq of R
end

types
  R ::
    size : real;

op(z:nat) r:real ==
(
    sv(1).size := 456;    -- PO #1
    return 1/sv(1).size   -- PO #2,3
);
```

```
Proof Obligation 1: (Unproved)
(forall z:nat, mk_Sigma(sv):Sigma &
  1 in set inds sv)

Proof Obligation 2: (Unproved)
(forall z:nat, mk_Sigma(sv):Sigma &
  (let sv : seq of R = (sv ++
       {1 |-> mu(sv(1), size |-> 456)}) in
     1 in set inds sv))

Proof Obligation 3: (Unproved)
(forall z:nat, mk_Sigma(sv):Sigma &
  (let sv : seq of R = (sv ++
       {1 |-> mu(sv(1), size |-> 456)}) in
     (sv(1).size) <> 0))
```

Proof Obligations (local dcl state)

- State can be local, using "dcl" statements in a block
- Blocks can nest, so state has multiple scopes, including module scope
- Local state restricted to its block, but *effects can escape*…

```
state Sigma of
    sv : nat
end

op(z:nat) r:real ==
(
    dcl a:nat := 0;
    a := a + 1;

    ( dcl b:nat := a + 1;
      sv := b );  <---- Updates state!

    ( dcl c:nat := a + 2;
      c := c + 1 );

    return 1/sv   -- PO#1 depends on a, b but not c
);
```

```
Proof Obligation 1: (Unproved)
(forall z:nat, mk_Sigma(sv):Sigma &
  (let a : nat = 0 in
    (let a : nat = (a + 1) in
      (let b : nat = (a + 1) in   <--- NOTE
        (let sv : real = b in
          sv <> 0)))))
```

Proof Obligations (alternative paths)

- Alternative paths to reach a PO generate **multiple POs**
- Path expansion is typically small (we hope!)

```
state Sigma of
    sv : real
end

op(a:nat) r:real ==
(
    if a > 0 then
    || (
      sv := 1,
      sv := sv * 2
    )
    else
      sv := 999;

    return 1/sv  -- PO#s 1,2,3
);
```

```
Proof Obligation 1: (Unproved)
(forall a:nat, mk_Sigma(sv):Sigma &
  ((a > 0) =>
    (let sv : real = 1 in
      (let sv : real = (sv * 2) in
        sv <> 0))))

Proof Obligation 2: (Unproved)
(forall a:nat, mk_Sigma(sv):Sigma &
  ((a > 0) =>
    (let sv : real = (sv * 2) in
      (let sv : real = 1 in
        sv <> 0))))

Proof Obligation 3: (Unproved)
(forall a:nat, mk_Sigma(sv):Sigma &
  (not (a > 0) =>
    (let sv : real = 999 in
      sv <> 0)))
```

Proof Obligations (ambiguous variables)

- Operation calls, give *ambiguous states* (and *Unchecked* POs)
- "pure" operations okay, and "ext wr" clauses help
- Note: returned values are *always* ambiguous

```
state Sigma of
    s : nat
end

op(z:nat) r:real ==
(
    op2(z);
    -- s := 999
    return 1/s  -- PO#1
);

op2(a:nat) ==
(
    ...
    s := f(a);
    ...
)
```

```
Proof Obligation 1: (Unchecked)
(forall z:nat, mk_Sigma(s):Sigma &
  (-- Ambiguous operation call to op2, affects (s)? at 8:5
    s <> 0))
```

```
Proof Obligation 1: (Unproved)
(forall z:nat, mk_Sigma(s):Sigma &
  (-- Ambiguous operation call to op2, affects (s)? at 8:5
    (let s : nat = 999 in    <-- NOTE
      (-- Resolved ambiguity (s) at 9:5
        s <> 0))))
```

Proof Obligations (atomic updates)

- Atomic statements assign to variables simultaneously
- Any state invariant is only checked at the end
- Note the ***mk_Sigma!*** maximal type used in the check

```
state Sigma of
    sv : real
    xv : real
inv s == s.sv <> s.xv
end

op(a:nat) ==
(
    atomic   -- PO#1
    (
        sv := xv;
        xv := sv
    )
);
```

```
Proof Obligation 1: (Unproved)
(forall a:nat, mk_Sigma(sv, xv):Sigma &
  (let $atomic1 : real = xv in
    (let $atomic2 : real = sv in
      (let sv : real = $atomic1 in
        (let xv : real = $atomic2 in
          let s = mk_Sigma!(sv, xv) in ((s.sv) <> (s.xv)))))))
```

Proof Obligations (postconditions)

- Postconditions can refer to *RESULT* and "old" state
- Old state saved at the start; renamed to use "$" rather than "~"
- Return statements create *RESULT* if necessary

```
state Sigma of
    sv : real
    xv : real
end

op: nat ==> real
op(a) ==
(
    sv := a + 1;
    xv := sv + a;
    return xv
)
post RESULT > xv~ + sv~;
```

```
Proof Obligation 1: (Unproved)
(forall a:nat, mk_Sigma(sv, xv):Sigma &
  (let xv$ = xv, sv$ = sv in          <-- Old state
    (let sv : nat = (a + 1) in
      (let xv : nat = (sv + a) in
        (let RESULT = xv in           <-- Return xv
          (RESULT > (xv$ + sv$)))))))
```

Proof Obligations (loops)

- Loops can have *@LoopInvarant(exp)* annotations
- Inline functions help, but still difficult with more complex cases

```
state Sigma of
    s : seq of int
end

op(data:seq of int) ==
(
    dcl count : int := 0;
    s := data;

    -- @LoopInvariant(count + len s = len data);
    while s <> [] do
    (
        s := tl s;
        count := count + 1
    )

    -- Here, invariant holds and s = []
    ...
);
```

```
Proof Obligation 1: (Unproved)
(forall data:seq of int, mk_Sigma(s):Sigma &
  let body: seq of int * int +> seq of int * int
      body(s, count) ==
        (let s : seq of int = (tl s) in
          (let count : int = (count + 1) in
            mk_(s, count))),

      invariant: seq of int * int * seq of int +> bool
      invariant(s, count, data) ==
        ((count + (len s)) = (len data)),

      loop: seq of int * int * seq of int +> bool
      loop(s, count, data) ==
        s <> [] =>
          invariant(s, count, data) and
          let mk_(s, count) = body(s, count) in
            invariant(s, count, data)
            and loop(s, count, data)
  in
    (let count : int = 0 in
      (let s : seq of int = data in
        (loop(s, count, data)))))
```

Proof Obligations (limitations (1/2))

- This is VDM-SL mainly (can cope with simple VDM++)
  - `forall z:nat, obj_A(sv |-> sv):A & pre_op(z, new A(sv)) => ??`
  - How to construct "new" state for pre_op?
  - What about static instance variables? Threads/sync?? Bus/CPUs???
  - So complex VDM++/RT cases marked as "Unchecked"
- Operation calls assumed to make *everything* ambiguous
  - Unless they are "pure" or intra-module with "ext wr" clauses
  - Eliminating ambiguities needs multi-module, multi-state analysis
  - Operation calls can recurse, so need *measures*
- Cannot handle *always, trap* and *tixe* exception related statements
  - Not sure what conditions lead to exceptions (extra paths?)
  - Currently, related POs marked as "Unchecked"
- Variable hiding *in the source specification* can confuse POG
  - POG already detects some cases and marks POs as "Unchecked"
  - Detection is fairly easy; avoidance is harder

Proof Obligations (limitations (2/2))

- Loops need more work:
  - Loops with POs or alternative paths *within* the body?
  - Need *@LoopTermination* (like a *measure*)
  - Without a *@LoopInvariant* annotation:
    - Loops mark all *updated* variables as ambiguous
    - Any POs generated within the loop are "Unchecked"
- The POG itself should be verified:
  - Are POs complete (covering all obligations)?
  - Are POs correct (valid operation semantics for any path)?

- **Future Work**
  - Address these limitations!!

Proof Obligations (Try it out)

- The POG updates are in VDMJ on *GitHub*
  - See https://github.com/nickbattle/vdmj
  - *Release* page for 4.7.0-SNAPSHOT:
    - https://github.com/nickbattle/vdmj/releases/tag/4.7.0-1
    - Stable: *packaging-4.7.0-SNAPSHOT-distribution.zip*
    - Or use the VSIX: *vdm-vscode-1.5.1-patch.vsix*

Thank You!