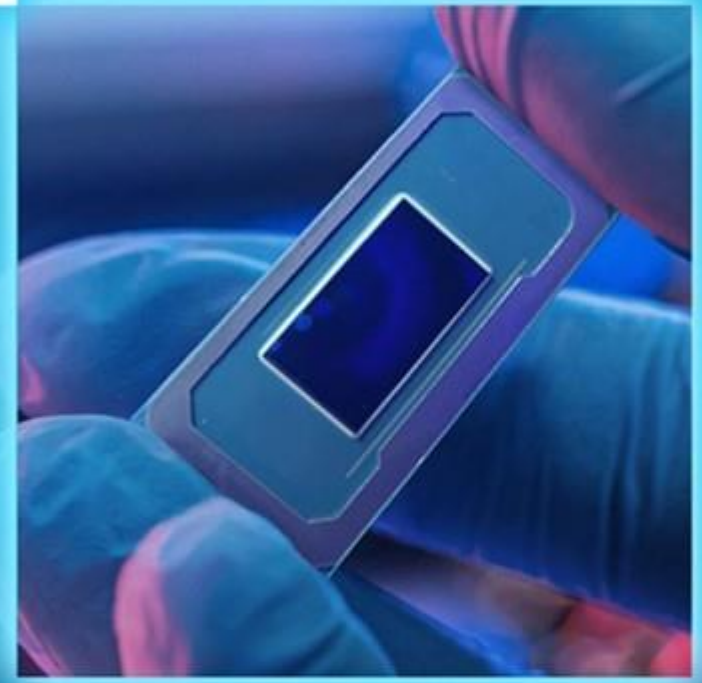
intel

Automating the search aspects of compiler engineering: IR for auto-tuning & beyond



Rolf Morel

<rolf.morel@intel.com>

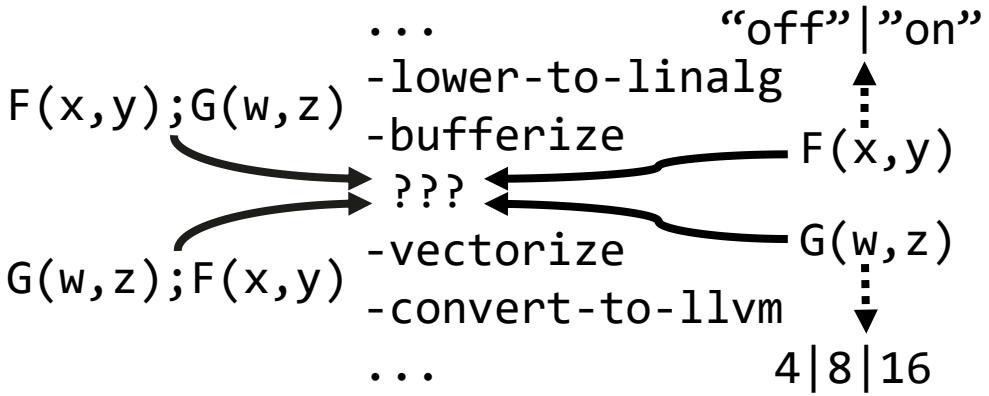
LLVM Dev Meeting – Oct 2025, Santa Clara

Premise:

Compiler engineering involves a lot tweaking of pipelines and knobs

An average plan for an average day:

1. Out of all possible optimizations, first try F. Evaluate. If perf isn't there...
2. Then tweak F's knobs. Evaluating each time.
3. Try optimization G. Evaluate.
4. Then tweak G's knobs. Evaluating each time.
5. Now try F;G. Tweak knobs. Evaluating each time.
6. Then G;F. Tweak knobs. Evaluating each time.

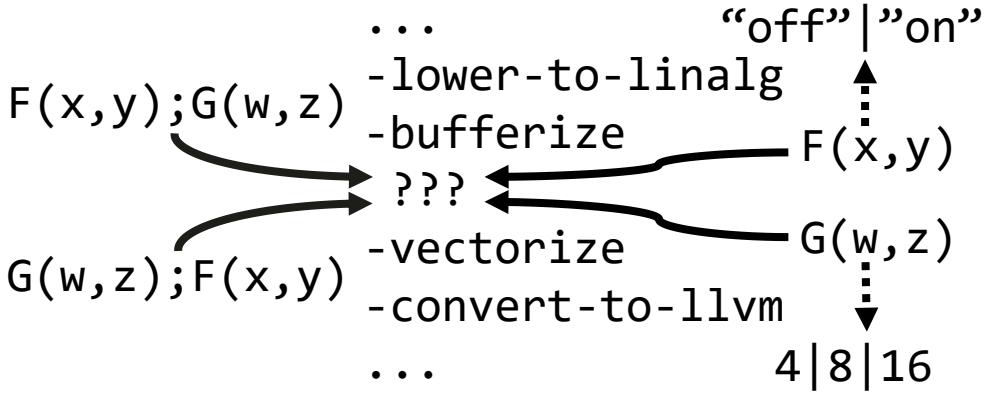


Promise:

Allow for ***expressing such plans*** and ***mechanize their execution***

An average plan for an average day:

1. Out of all possible optimizations, first try optimization F. Evaluate. If perf isn't there...
2. Then tweak F's knobs. Evaluating each time.
3. Try optimization G. Evaluate.
4. Then tweak G's knobs. Evaluating each time.
5. Now try F;G. Tweak knobs. Evaluating each time.
6. Then G;F. Tweak knobs. Evaluating each time.



In this talk,

expressing such plans via

- IR – an extension of *schedule* IR, i.e. pipelines as programs
  - which gives representation to the different choices we are considering

mechanize their execution via

- Mapping such schedules-with-choices to constraint problems
  - so, we can hand off the combinatorial search to solvers

Agenda

1. RECAP:
    I. **Schedules**: an alternative to pipelines
    II. Rewriting MLIR with its **Transform dialect**
2. IR for tuning knobs … with joint constraints
3. Non-deterministic semantics via SMT
    … and regaining determinism through rewrites
4. IR for alternative transform sequences
5. Mechanized schedule/pipeline optimization

RECAP:
What are schedules?

program = algorithm + schedule

Image processing à la Halide

algorithm

```
blurx(x,y) = in(x-1,y)
           + in(x,y)
           + in(x+1,y)
out(x,y) = blurx(x,y-1)
         + blurx(x,y)
         + blurx(x,y+1)
```

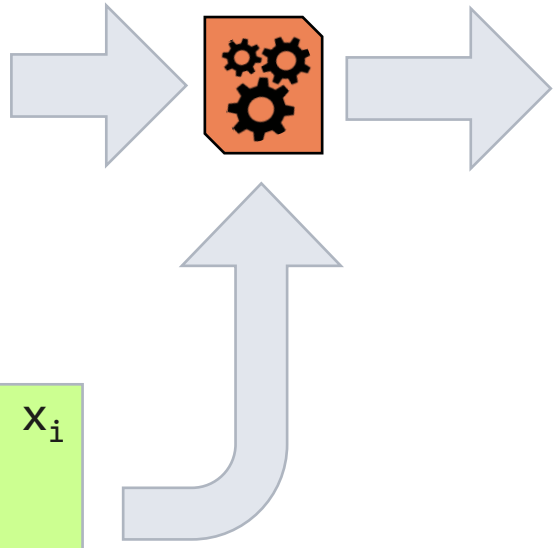
program

```
par for out.y_o in 0..out.y.extent/4
 for out.x_o in 0..out.x.extent/4
  alloc blurx[blurx.y.extent][blurx.x.extent]
  for out.y_i in 0..4
   let blurx.y.min = 4*out.y_o.min + out.y_i.min - 1
   for blurx.y in blurx.y.min..blurx.y.max
    for blurx.x_o in blurx.x.min/4..blurx.x.max/4
     vec for blurx.x_i in 0..4
      blurx[blurx.y.stride*blurx.y+...] =
          in[in.y.stride*(blurx.y.min+blurx.y)
             +4*blurx.x_o+ramp(4)] + ...
   vec for out.x_i in 0..4
    out[out.y.stride*(4*(out.y_o-out.y_o.min)+out.y_i)+...] =
      blurx[blurx.y.stride*(out.y_i-1-blurx.y.min)
            + out.x_i - blurx.x.min] + ...
```

Image processing à la Halide

algorithm (high-level code)

```
blurx(x,y) = in(x-1,y)
           + in(x,y)
           + in(x+1,y)
out(x,y) = blurx(x,y-1)
         + blurx(x,y)
         + blurx(x,y+1)
```



schedule

```
blurx: split x by 4 → x_o, x_i
       vectorize: x_i
       store at out.x_0
       compute at out.y_i
out: split x by 4 → x_o, x_i
     split y by 4 → y_o, y_i
     reorder: y_o, x_o, y_i, x_i
     parallelize: y_o
     vectorize: x_i
```

program (low-level & optimized code)

```
par for out.y_o in 0..out.y.extent/4
 for out.x_o in 0..out.x.extent/4
  alloc blurx[blurx.y.extent][blurx.x.extent]
  for out.y_i in 0..4
   let blurx.y.min = 4*out.y_o.min + out.y_i.min - 1
   for blurx.y in blurx.y.min..blurx.y.max
    for blurx.x_o in blurx.x.min/4..blurx.x.max/4
     vec for blurx.x_i in 0..4
      blurx[blurx.y.stride*blurx.y+...] =
          in[in.y.stride*(blurx.y.min+blurx.y)
             +4*blurx.x_o+ramp(4)] + ...
   vec for out.x_i in 0..4
    out[out.y.stride*(4*(out.y_o-out.y_o.min)+out.y_i)+...] =
      blurx[blurx.y.stride*(out.y_i-1-blurx.y.min)
            + out.x_i - blurx.x.min] + ...
```

Halide (Ragan-Kelley et al, 2013)

Many scheduling DSLs

LIFT & RISE
(Steuwer et al, 2015 & 2022)

Fireiron
(Hagedorn et al, 2020)

… in support of
optimizing BLAS / tensor programs:

```
A = t.placeholder((1024, 1024))
B = t.placeholder((1024, 1024))
k = t.reduce_axis((0, 1024))
C = t.compute((1024, 1024), lambda y, x:
                t.sum(A[k, y] * B[k, x], axis=k))
s = t.create_schedule(C.op)
```

```
for y in range(1024):
  for x in range(1024):
    C[y][x] = 0
    for k in range(1024):
      C[y][x] += A[k][y] * B[k][x]
```

+ Loop Tiling

```
yo, xo, ko, yi, xi, ki = s[C].tile(y, x, k, 8, 8, 8)
```

```
for yo in range(128):
  for xo in range(128):
    C[yo*8:yo*8+8][xo*8:xo*8+8] = 0
    for ko in range(128):
      for yi in range(8):
        for xi in range(8):
          for ki in range(8):
            C[yo*8+yi][xo*8+xi] +=
              A[ko*8+ki][yo*8+yi] * B[ko*8+ki][xo*8+xi]
```

TVM (Chen et al, 2018)

Exo
(Ikarashi et al, 2022)

Tiramisu
(Baghdadi et al, 2019)

All share the same *core idea*:
declarative descriptions of how to transform code

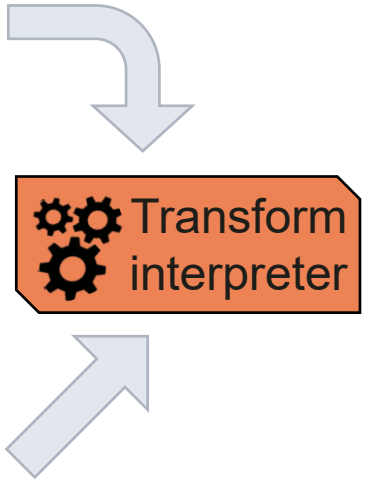
RECAP:
What is MLIR's Transform Dialect?

MLIR's Transform *meta*-Dialect

transform IR describes transformations on ***payload IR***

payload IR:

```
.mlir
func.func @gemv(%α, %A, %x, %β, %y) {
 %βy = linalg.generic attrs { iter_types = ["par"] } ... {
  %βy_elem = arith.mulf %β_elem, %y_elem
  linalg.yield %βy_elem : f32
 } -> tensor<?xf32>
 ...
 %αAx_plus_βy = linalg.generic
   { iter_types = ["par", "reduction"] } ... {
    ...
 } -> tensor<?x?xf32>
 func.return %αAx_plus_βy : tensor<?x?xf32>
}
```



transform IR:

```
.mlir
transform.sequence(%func: !transform.any_op) {
  %elemwise = transform.match attrs { iter_types = ["par"] } %func
  transform.tile_using_for %elemwise [4]
}
```

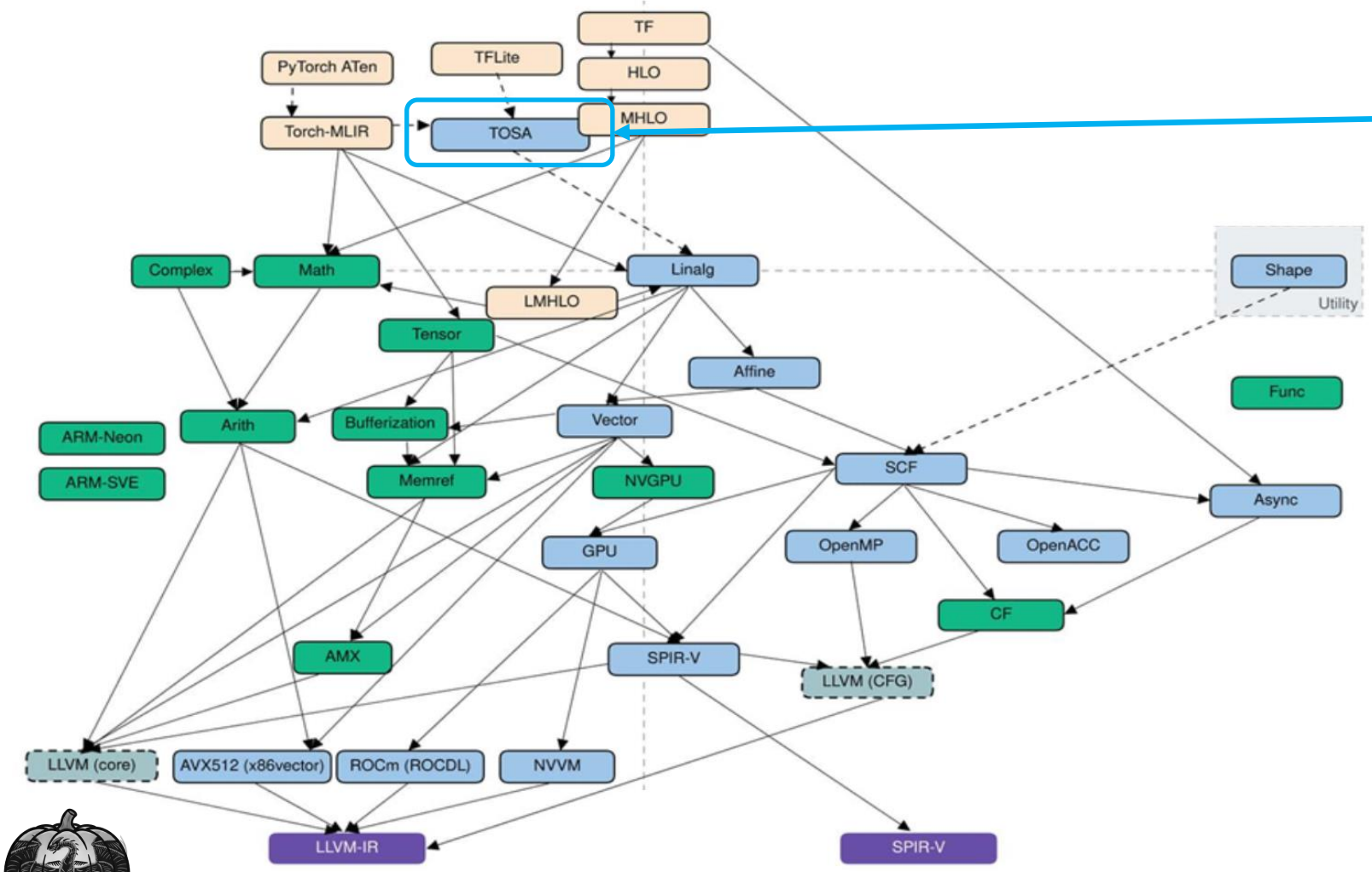
Transformed payload IR:

```
.mlir
func.func @gemv(%α, %A, %x, %β, %y) {
 %dim = tensor.dim %y, %c0
 %βy = scf.for %i = %c0 to %dim step %c4 ... {
  %ex_slice = tensor.extract_slice ...
  ... = linalg.generic attrs {iter_types = ["par"]} {
   %βy_slice_elem = arith.mulf %β_elem, %y_slice_elem
   linalg.yield %βy_slice_elem : f32
  } -> tensor<?xf32>
  %in_slice = tensor.insert_slice ...
  scf.yield %in_slice
 }
 ...
 %αAx_plus_βy = linalg.generic
   { iter_types = ["par", "reduction"]} ... {
    ...
 } -> tensor<?x?xf32>
 func.return %αAx_plus_βy : tensor<?x?xf32>
}
```

The MLIR Transform Dialect: Your Compiler Is More Powerful Than You Think

(Martin Paul Lücke, Alex Zinenko, William S. Moses, Michel Steuwer, Albert Cohen)

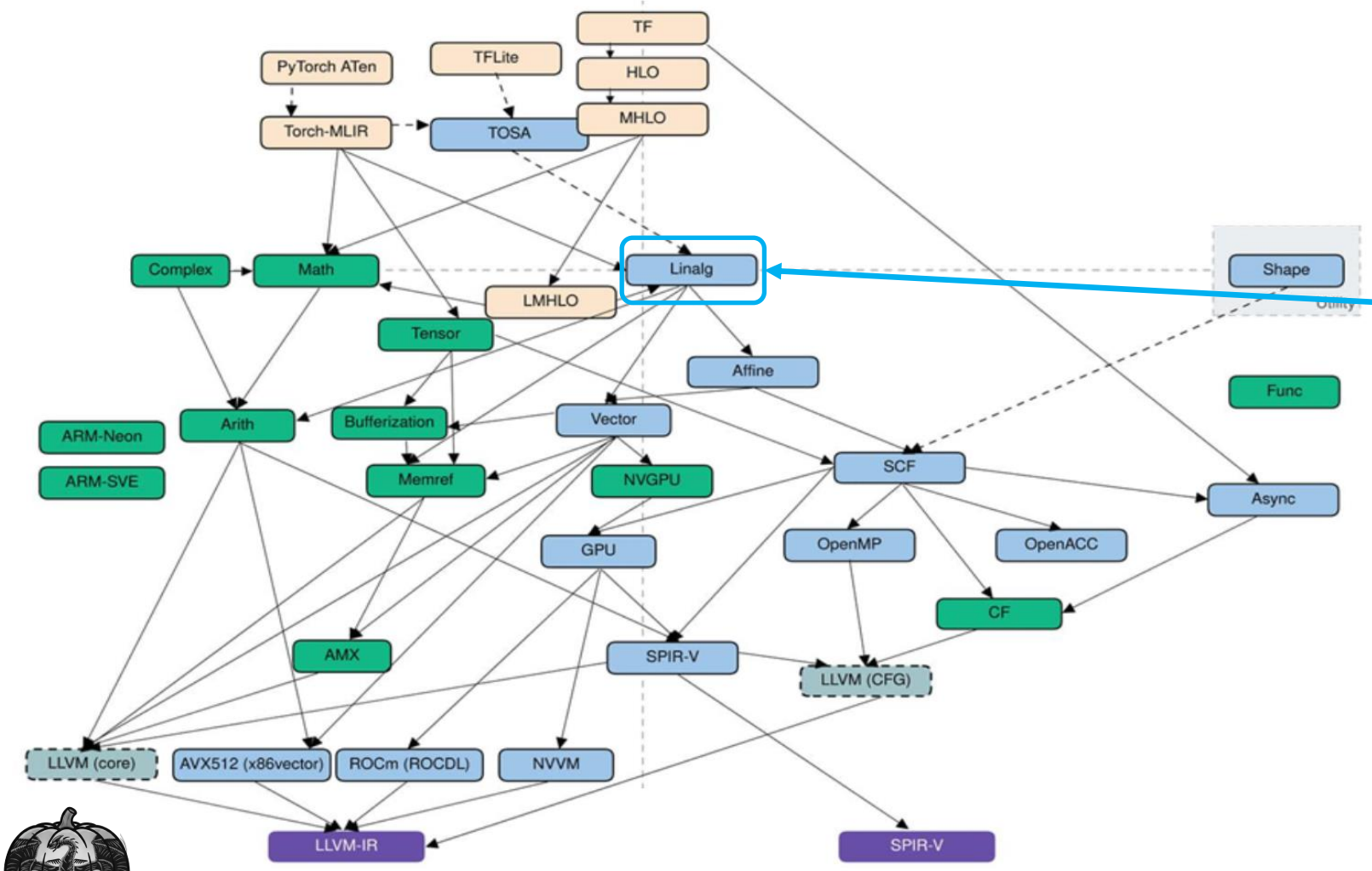
Progressive lowering/optimization
through the dialects

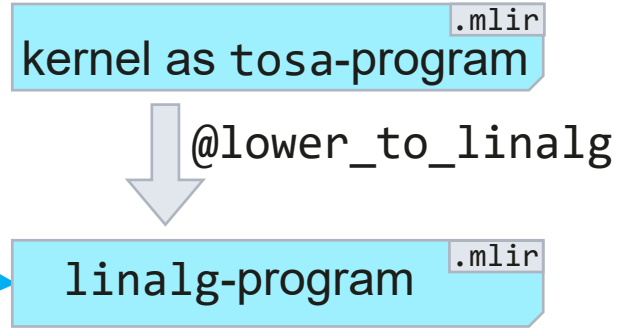


.mlir
kernel as `tosa-program`

(image credit: Alex Zinenko & Quinn Dawkins)

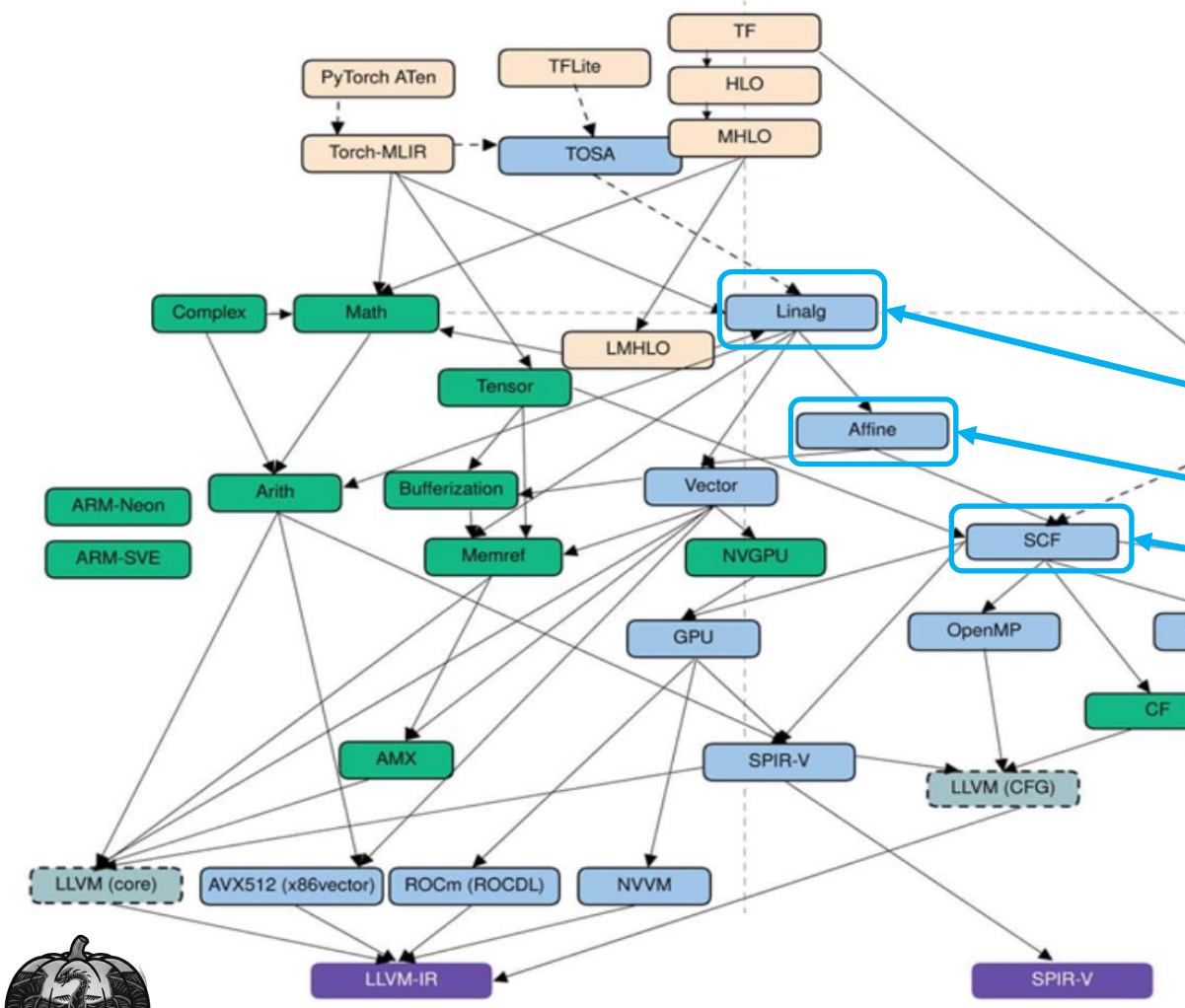
Progressive lowering/optimization
through the dialects

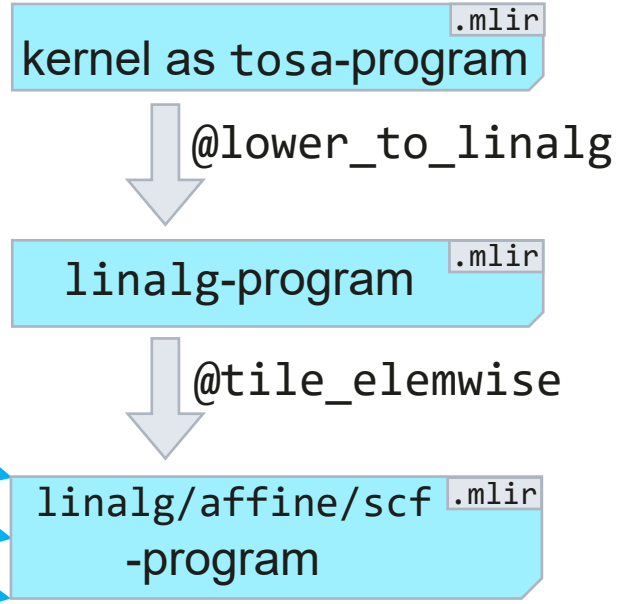




(image credit: Alex Zinenko & Quinn Dawkins)

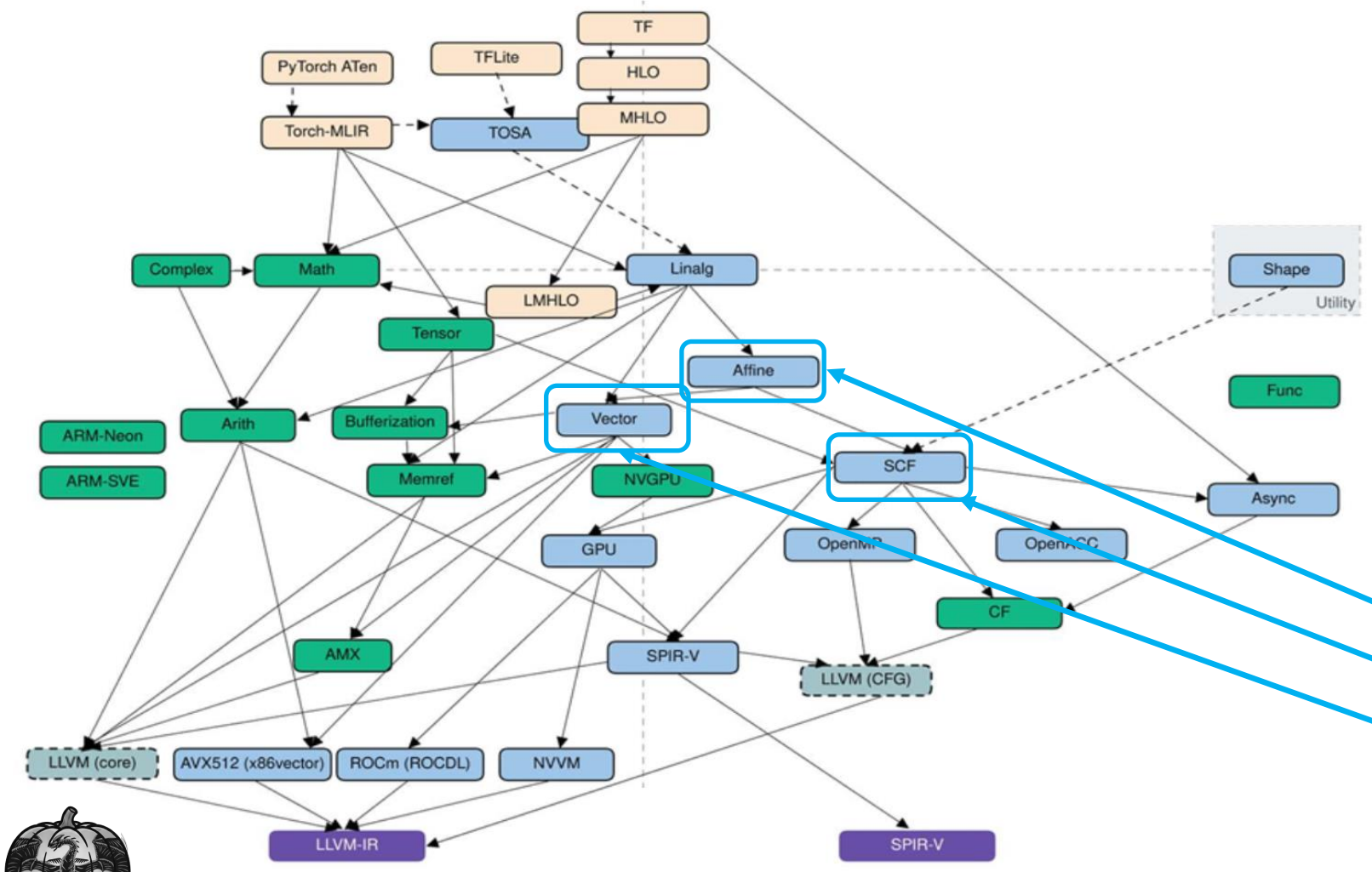
Progressive lowering/optimization
through the dialects

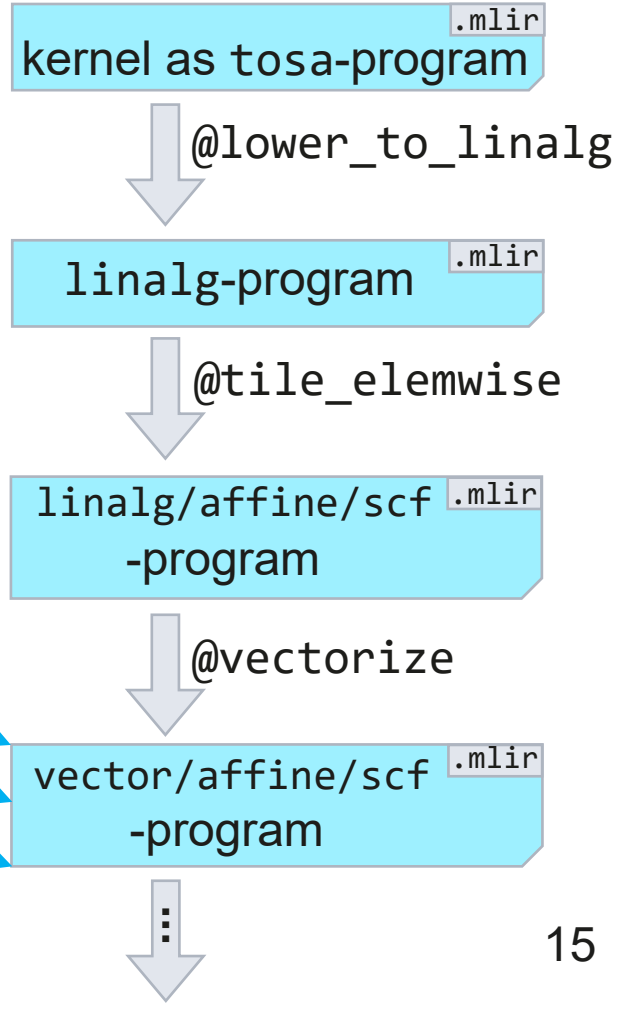




(image credit: Alex Zinenko & Quinn Dawkins)

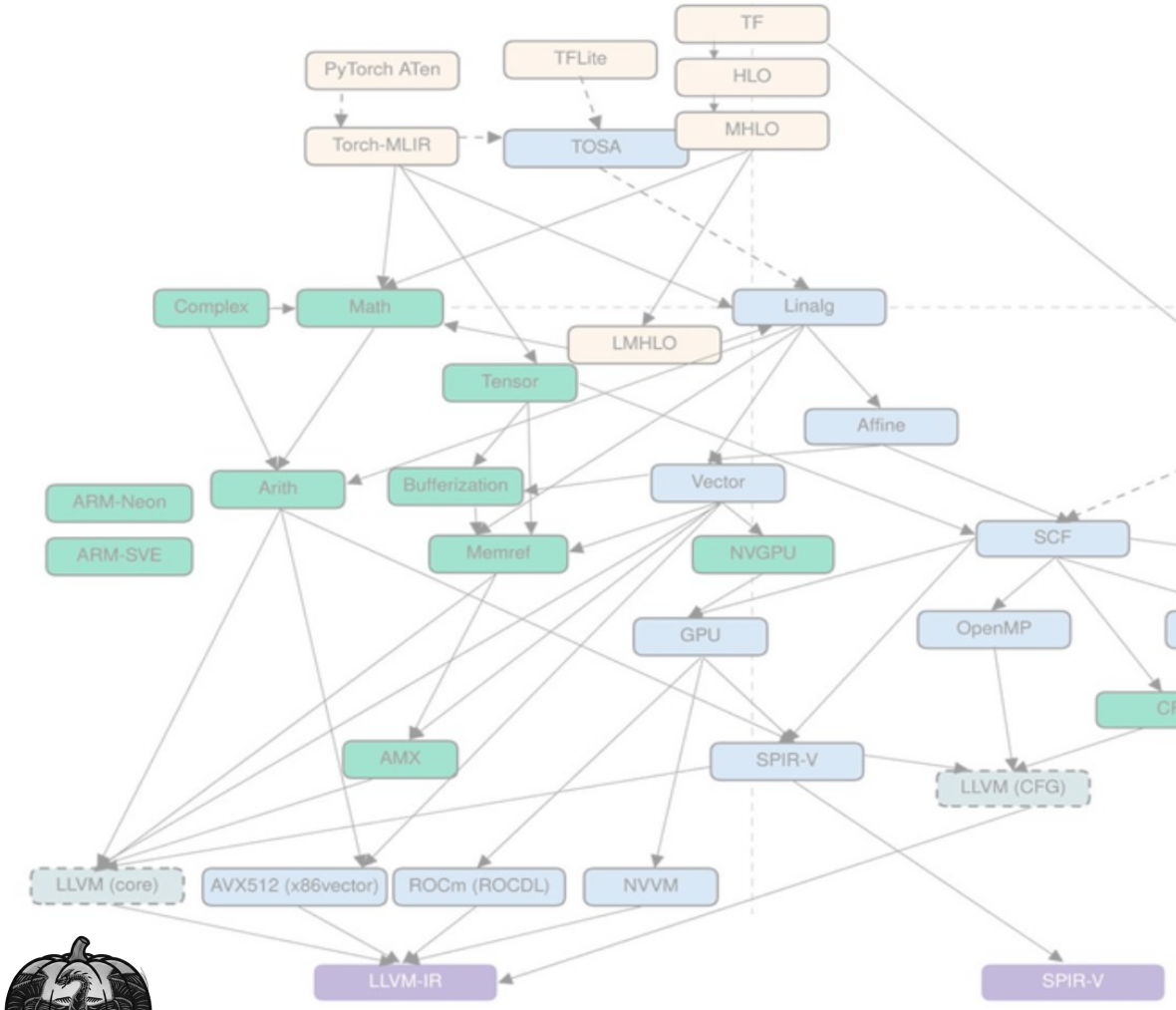
Progressive lowering/optimization
through the dialects





(image credit: Alex Zinenko & Quinn Dawkins)

Progressive lowering/optimization
through the dialects, schedule by schedule




```
// @lower_to_linalg
...
transform.apply_registered_pass
  "tosa-to-linalg-pipeline" to %0
...
```
.mlir

```
// @tile_elemwise
...
transform.tile_using_for %0 [4]
...
```
.mlir

```
// @vectorize
...
transform.vectorize %0 [4, 8]
...
```
.mlir

kernel as tosa-program .mlir

Transform interpreter

linalg-program .mlir

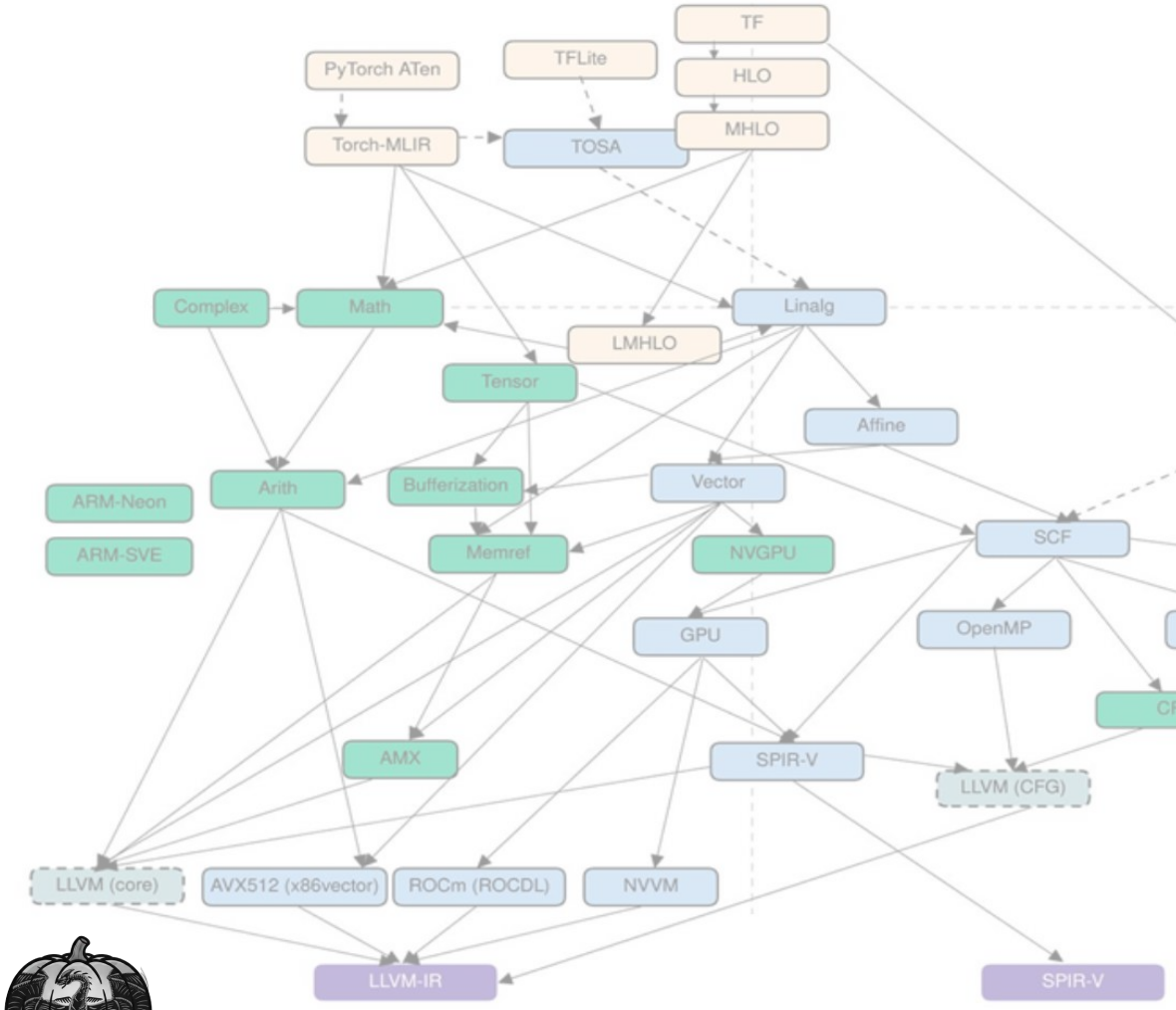
Transform interpreter

linalg/affine/scf -program .mlir

Transform interpreter

vector/affine/scf -program .mlir

Progressive lowering/optimization
through the dialects, via an overall schedule




```
.mlir
transform.sequence {
 // @lower_to_linalg
 ...
 %1 = transform.apply_registered_pass
   "tosa-to-linalg-pipeline" to %0
 ...
 // @tile_outer_dims
 ...
 %2 = transform.tile_using_for %1 [4]
 ...
 // @vectorize
 ...
 %3 = transform.vectorize %2 [4, 8]
 ...
}
```

.mlir
kernel as tosa-program

Transform interpreter

.mlir
vector/affine/scf -program

What about tuning schedules?

Tuning schedules: selecting tile sizes

```
transform.sequence(%mod) {
  ...
  %target = transform.match “linalg.matmul” %mod
  %tiled, %cache_loops:2 = transform.tile_using_for tile_sizes [16,32] %target
  ...
  %tiled2, %reg_loops:2 = transform.tile_using_for tile_sizes [4,6] %tiled
  ...
}
```
.mlir

Tuning schedules: selecting tile sizes

```
transform.sequence(%mod) {
  ...
  %target = transform.match “linalg.matmul” %mod
  %tiled, %cache_loops:2 = transform.tile_using_for tile_sizes [32,32] %target
  ...
  %tiled2, %reg_loops:2 = transform.tile_using_for tile_sizes [4,6] %tiled
  ...
}
```
.mlir

Tuning schedules: selecting tile sizes

```
transform.sequence(%mod) {
  ...
  %target = transform.match “linalg.matmul” %mod
  %tiled, %cache_loops:2 = transform.tile_using_for tile_sizes [32,64] %target
  ...
  %tiled2, %reg_loops:2 = transform.tile_using_for tile_sizes [5,5] %tiled
  ...
}
```
.mlir

Schedules with choices: "*reifying*" knobs

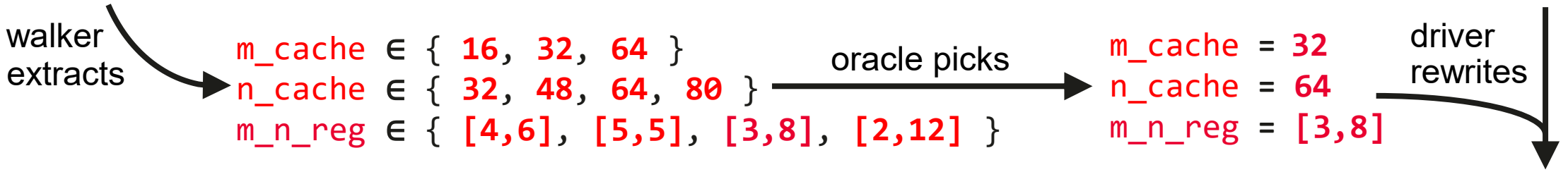
```
.mlir
transform.sequence(%mod) {
  ...
  %target = transform.match "linalg.matmul" %mod
  %m_cache = transform.tune.knob<"m_cache"> options = [16, 32, 64]
  %n_cache = transform.tune.knob<"n_cache"> options = [32, 48, 64, 80]
  %tiled, %cache_loops:2 = transform.tile_using_for tile_sizes [%m_cache,%n_cache] %target
  ...
  %m_n_reg = transform.tune.knob<"m_n_reg"> options = [[4,6],[5,5],[3,8],[2,12]]
  %m_reg, %n_reg = transform.param.split_handle %m_n_reg
  %tiled2, %reg_loops:2 = transform.tile_using_for tile_sizes [%m_reg,%n_reg] %tiled
  ...
}
```

From which we can automatically extract the parameter space:

$$
\begin{aligned}
\texttt{m\_cache} &\in \{\ 16,\ 32,\ 64\ \} \\
\texttt{n\_cache} &\in \{\ 32,\ 48,\ 64,\ 80\ \} \\
\texttt{m\_n\_reg} &\in \{\ [4,6],\ [5,5],\ [3,8],\ [2,12]\ \}
\end{aligned}
$$

Rewriting knobs: regaining determinism

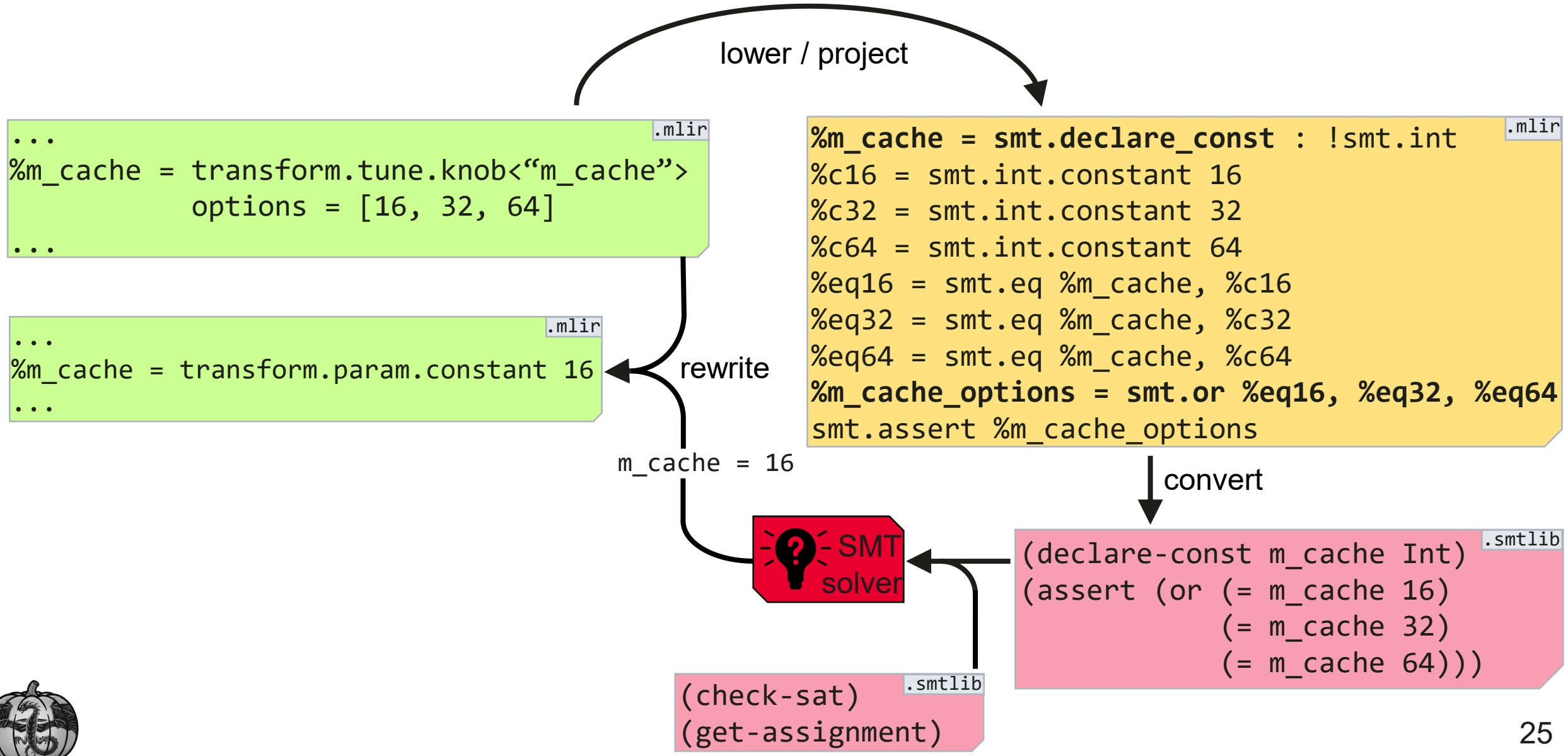
```
.mlir
%m_cache = transform.tune.knob<"m_cache"> options = [16, 32, 64]
%n_cache = transform.tune.knob<"n_cache"> options = [32, 48, 64, 80]
%tiled, %cache_loops:2 = transform.tile_using_for tile_sizes [%m_cache,%n_cache] %target
...
%m_n_reg = transform.tune.knob<"m_n_reg"> options = [[4,6],[5,5],[3,8],[2,12]]
%m_reg, %n_reg = transform.param.split_handle %m_n_reg
%tiled2, %reg_loops:2 = transform.tile_using_for tile_sizes [%m_reg,%n_reg] %tiled
```



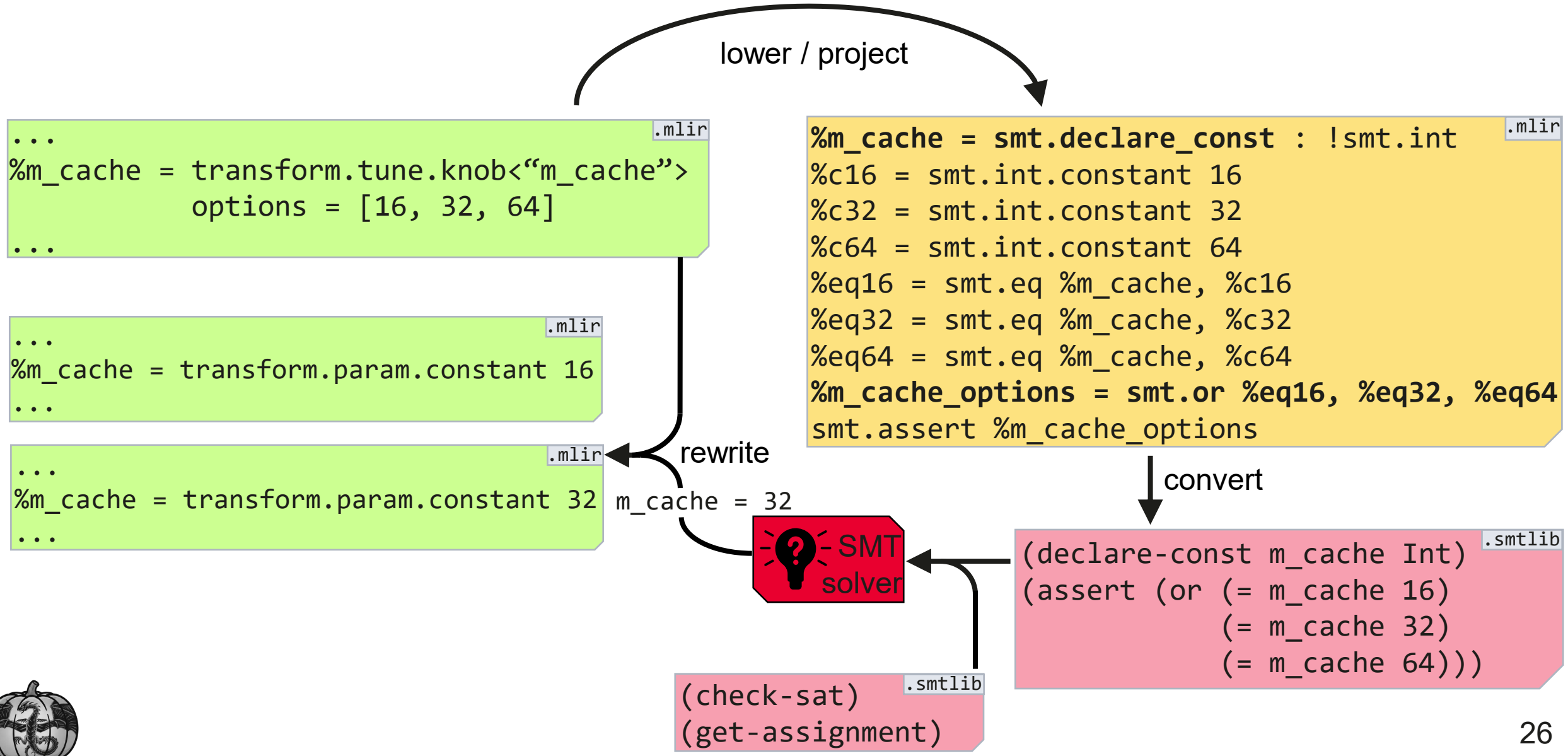

```
.mlir
%m_cache = transform.param.constant 32
%n_cache = transform.param.constant 64
%tiled, %cache_loops:2 = transform.tile_using_for tile_sizes [%m_cache,%n_cache] %target
...
%m_n_reg = transform.param.constant [3,8]
%m_reg, %n_reg = transform.param.split_handle %m_n_reg
%tiled2, %reg_loops:2 = transform.tile_using_for tile_sizes [%m_reg,%n_reg] %tiled
```

Non-deterministic semantics via SMT

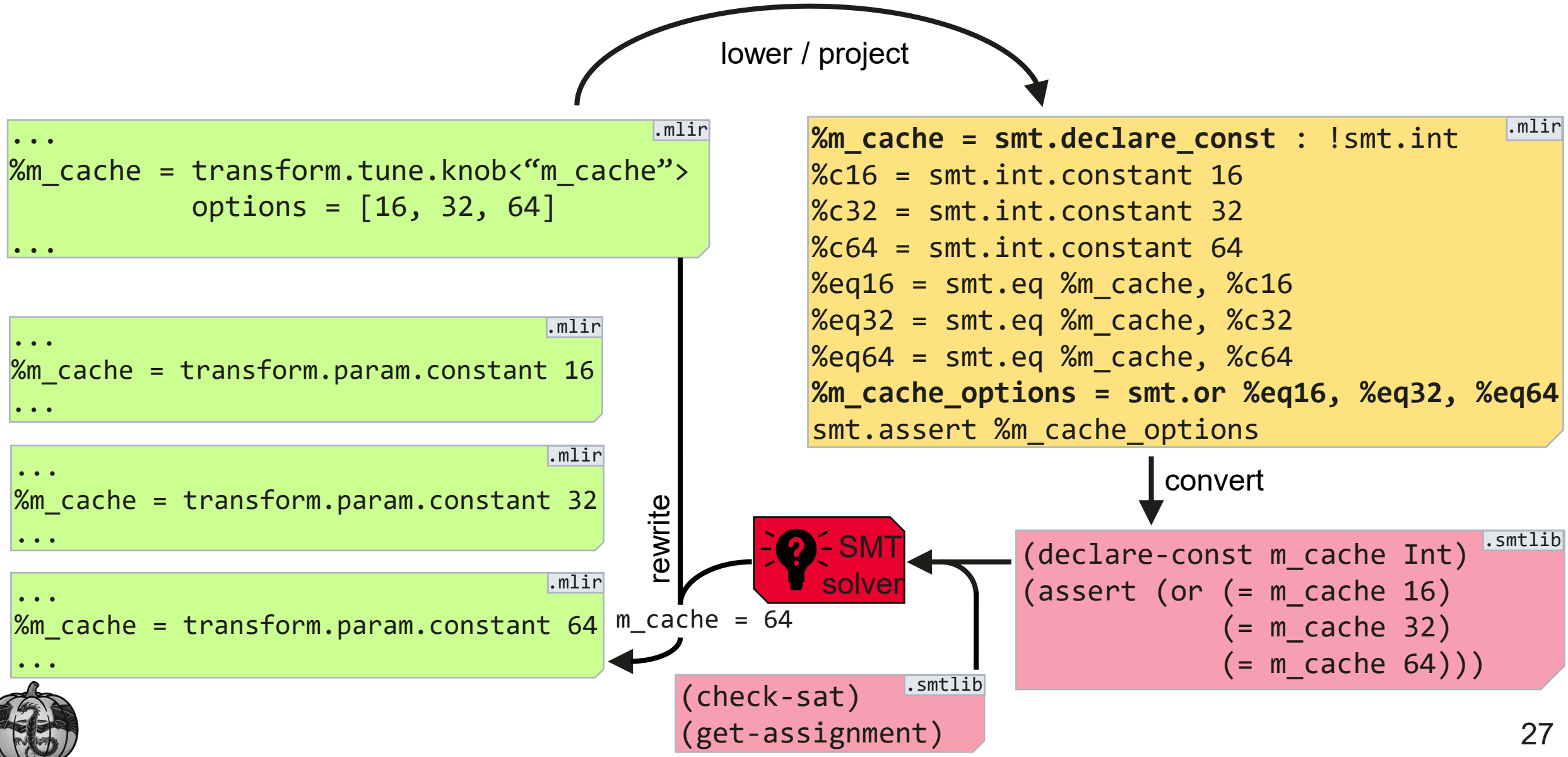
Non-deterministic semantics via SMT-dialect



Non-deterministic semantics via SMT-dialect



Non-deterministic semantics via SMT-dialect



Tuning schedules: joint choices via SMT

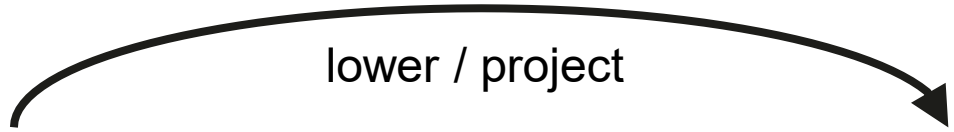
Can hand-code the known valid combinations:

```mlir
...
%m_n_reg = transform.tune.knob<“m_n_reg”> options = [[4,6],[5,5],[3,8],[2,12]]
%m_reg, %n_reg = transform.param.split_handle %m_n_reg
...
```

Or can express actual constraint, e.g. m_{reg} * n_{reg} ≤ 25, over independent variables:

```mlir
...
%m_reg = transform.tune.knob<“m_reg”> options = range<2,12> : !transform.param<i64>
%n_reg = transform.tune.knob<“n_reg”> options = range<2,12> : !transform.param<i64>
transform.smt.constrain_params(%m_reg, %n_reg) {
  ^bb(%m_var: !smt.int, %n_var: !smt.int):
  %m_times_n = smt.int.mul %m_var, %n_var
  %c25 = smt.int.constant 25
  %prod_le_25 = smt.int.cmp le %m_times_n, %c25
  smt.assert %prod_le_25
}
...
```

Tuning schedules: joint choices as SMT



.mlir

```
%m_reg = transform.tune.knob<“m_reg”>
  options = range<2,12> : !transform.param<i64>
%n_reg = transform.tune.knob<“n_reg”>
  options = range<2,12> : !transform.param<i64>
transform.smt.constrain_params(%m_reg, %n_reg) {
  ^bb(%m_var: !smt.int, %n_var: !smt.int):
  %m_times_n = smt.int.mul %m_var, %n_var
  %c25 = smt.int.constant 25
  %prod_le_25 = smt.int.cmp le %m_times_n %c25
  smt.assert %prod_le_25
}
```

.mlir

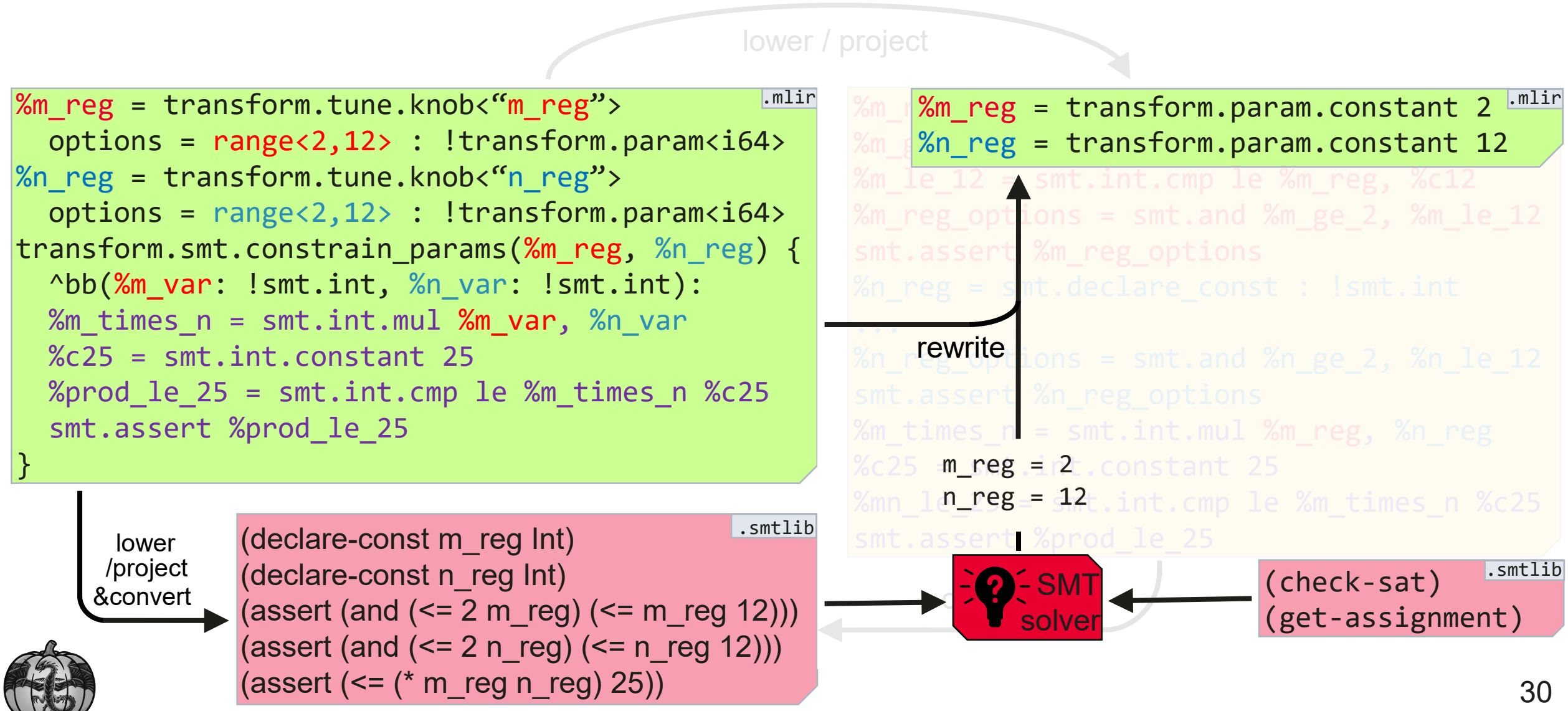
```
%m_reg = smt.declare_const : !smt.int
%m_ge_2 = smt.int.cmp le %c2, %m_reg
%m_le_12 = smt.int.cmp le %m_reg, %c12
%m_reg_options = smt.and %m_ge_2, %m_le_12
smt.assert %m_reg_options
%n_reg = smt.declare_const : !smt.int
...
%n_reg_options = smt.and %n_ge_2, %n_le_12
smt.assert %n_reg_options
%m_times_n = smt.int.mul %m_reg, %n_reg
%c25 = smt.int.constant 25
%mn_le_25 = smt.int.cmp le %m_times_n %c25
smt.assert %prod_le_25
```

convert

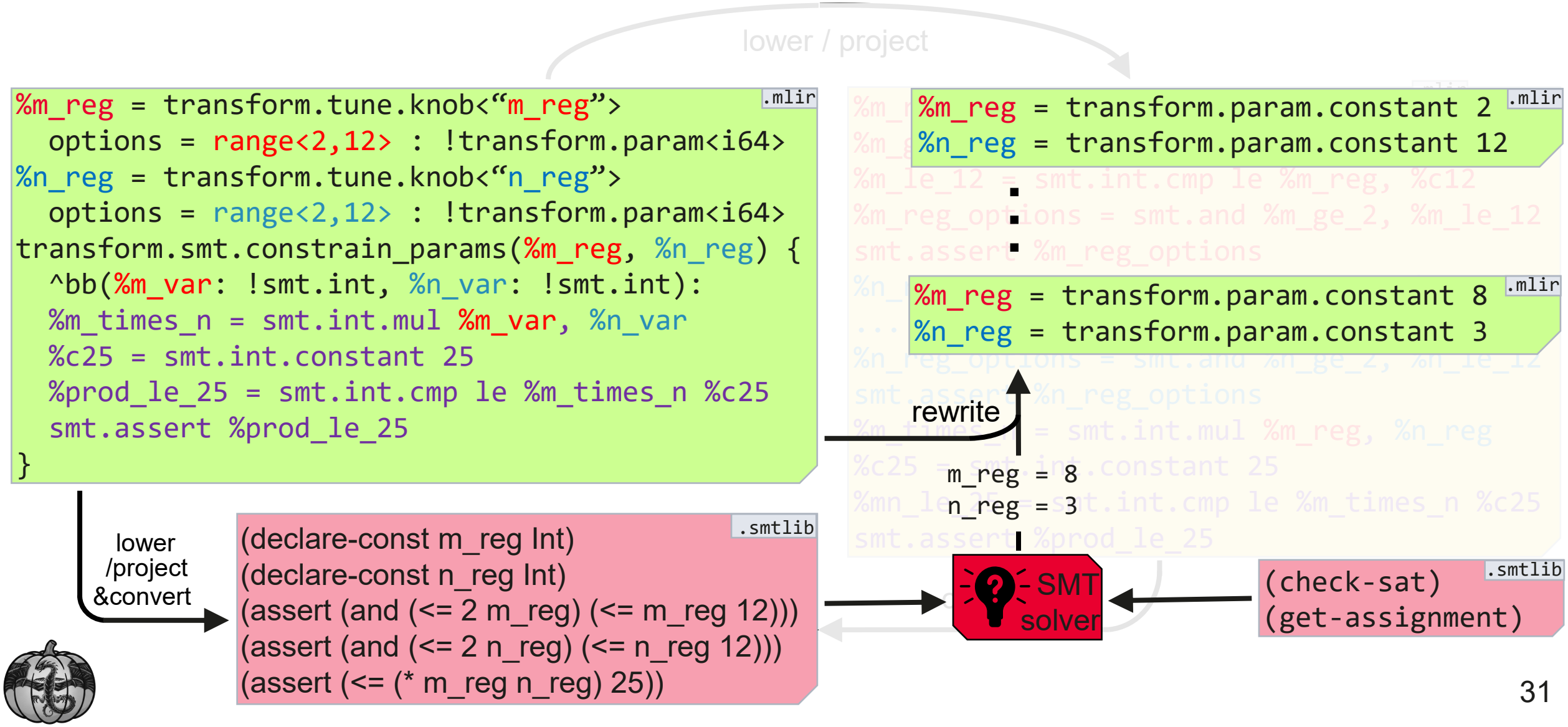
.smtlib

```
(declare-const m_reg Int)
(declare-const n_reg Int)
(assert (and (<= 2 m_reg) (<= m_reg 12)))
(assert (and (<= 2 n_reg) (<= n_reg 12)))
(assert (<= (* m_reg n_reg) 25))
```

Tuning schedules: joint choices as SMT



Tuning schedules: joint choices as SMT



Beyond tuning knobs: alternative transform sequences

Choice among transforms: parallel or seq

```
…A…
%tiled, %forall_loop =
  transform.tile_using_forall
    tile_sizes [%m,%n] %target
…B…
```
.mlir

or

```
…A…
%tiled, %for_loops:2 =
  transform.tile_using_for
    tile_sizes [%m,%n] %target
…B…
```
.mlir

Parameter/knob tuning does *not* allow for
expressing choice between different transforms!

Choice among transforms: parallel or seq

```
…A…
%tiled, %forall_loop =
  transform.tile_using_forall
    tile_sizes [%m,%n] %target
…B…
```
.mlir

or

```
…A…
%tiled, %for_loops:2 =
  transform.tile_using_for
    tile_sizes [%m,%n] %target
…B…
```
.mlir

can be represented by

```
…A…
%tiled_target, %loop = transform.tune.alternatives<“par_or_seq”> {
  %tiled, %forall_loop = transform.tile_using_forall tile_sizes [%m,%n] %target
  transform.yield %tiled, %forall_loop
}, {
  %tiled, %for_loops:2 = transform.tile_using_for tile_sizes [%m,%n] %target
  transform.yield %tiled, %for_loops#0
}
…B…
```
.mlir

which rewrites to

```
…A…
%tiled_target, %loop =
  transform.tile_using_forall
    tile_sizes [%m,%n] %target
…B…
```
.mlir

or

```
…A…
%tiled_target, %loop, %_ =
  transform.tile_using_for
    tile_sizes [%m,%n] %target
…B…
```
.mlir

Sub-schedule choice: mapping to SMT

.mlir

```
%res = transform.tune.alternatives<“subsched”> {
  …subsched0…
  transform.yield %subsched0_result
}, {
  …subsched1…
  transform.yield %subsched1_result
}, {
  ...
}, {
  …subschedN…
  transform.yield %subschedN_result
}
```

.mlir

```
%subsched = smt.declare_const : !smt.int
%subsched_lb = smt.int.cmp le %c0, %subsched
%subsched_ub = smt.int.cmp le %subsched, %cN
%subsched_options = smt.and %subsched_lb,
                            %subsched_ub
smt.assert %subsched_options

%subsched_region0 = smt.eq %subsched, %c0
// SMT-ops for subsched0
// … conditional on %subsched == 0

%subsched_region1 = smt.eq %subsched, %c1
// SMT-ops for subsched1
// … conditional on %subsched == 1
...
% subsched_regionN = smt.eq %subsched, %cN
// SMT-ops for subschedN
// … conditional on %subsched == N
```

Sub-schedule choice: path dependence

.mlir

```
%tiled_target, %loop = transform.tune.alternatives<“par_or_seq”> {
  %x = transform.tune.knob<“x”> options = [4,8,16]
  %tiled, %forall_loop = transform.tile_using_forall tile_sizes [%x,4] %target
  transform.yield %tiled, %forall_loop
}, {
  %y = transform.tune.knob<“y”> options = [2,4]
  %tiled, %for_loops:2 = transform.tile_using_for tile_sizes [8,%y] %target
  transform.yield %tiled, %for_loops#0
}
```

lower /project

.mlir

```
%x = smt.declare_const : !smt.int
...
%x_options = smt.or %x_eq_4, %x_eq_8, %x_eq_16


smt.assert %x_options
%y = smt.declare_const : !smt.int
...
%y_options = smt.or %y_eq_2, %y_eq_4


smt.assert %y_options
```

Sub-schedule choice: path dependence

```
.mlir
%tiled_target, %loop = transform.tune.alternatives<“par_or_seq”> {
  %x = transform.tune.knob<“x”> options = [4,8,16]
  %tiled, %forall_loop = transform.tile_using_forall tile_sizes [%x,4] %target
  transform.yield %tiled, %forall_loop
}, {
  %y = transform.tune.knob<“y”> options = [2,4]
  %tiled, %for_loops:2 = transform.tile_using_for tile_sizes [8,%y] %target
  transform.yield %tiled, %for_loops#0
}
```

lower /project

```
.mlir
%par_or_seq = smt.declare_const : !smt.int
%lb = smt.int.cmp ge %par_or_seq, %c0
%ub = smt.int.cmp le %par_or_seq, %c1
%par_or_seq_options = smt.and %lb, %ub
smt.assert %par_or_seq_options

%par_or_seq_region0 = smt.eq %par_or_seq, %c0

%par_or_seq_region1 = smt.eq %par_or_seq, %c1
```

```
%x = smt.declare_const : !smt.int
...
%x_options = smt.or %x_eq_4, %x_eq_8, %x_eq_16
%conditional_x_options = smt.implies %par_or_seq_region0,
                                     %x_options
smt.assert ~~%x_options~~ %conditional_x_options
%y = smt.declare_const : !smt.int
...
%y_options = smt.or %y_eq_2, %y_eq_4
%conditional_y_options = smt.implies %par_or_seq_region1,
                                     %y_options
smt.assert ~~%y_options~~ %conditional_y_options
```

Tied choices across sub-schedules

```mlir
%tiled_target, %forall_loop =
  transform.tile_using_forall
    tile_sizes [4,8] %target
...
// do something with %tiled_target
...
transform.loop.forall_to_parallel %forall_loop
```

or

```mlir
%tiled_target, %loops:2 =
  transform.tile_using_for
    tile_sizes [4,8] %target
...
// do same something with %tiled_target
...
transform.loop.coalesce %loops#0
```

Tied choices across sub-schedules

```
.mlir
%marked_par = transform.param.constant 1
%marked_seq = transform.param.constant 0
%tiled_target, %loop, %marker = transform.tune.alternatives<“par_or_seq”> {
  %tiled, %forall_loop = transform.tile_using_forall tile_sizes [4,8] %target
  transform.yield %tiled, %forall_loop, %marked_par
}, {
  %tiled, %for_loops:2 = transform.tile_using_for tile_sizes [4,8] %target
  transform.yield %tiled, %for_loops#0, %marked_seq
}
...
// do something with %tiled_target
...
transform.tune.alternatives<“par_or_seq_reprise”> {
  transform.param.cmpi eq %marker, %marked_par  // Crash if not equal!!!
  transform.loop.forall_to_parallel %loop
  transform.yield
}, {
  transform.param.cmpi eq %marker, %marked_seq // Crash if not equal!!!
  transform.loop.coalesce %loop
  transform.yield
}
```

Tied choices across sub-schedules

```
%marked_par = transform.param.constant 1
%marked_seq = transform.param.constant 0
%tiled_target, %loop, %marker = transform.tune.alternatives<“par_or_seq”> {
  %tiled, %forall_loop = transform.tile_using_forall tile_sizes [4,8] %target
  transform.yield %tiled, %forall_loop, %marked_par
}, {
  %tiled, %for_loops:2 = transform.tile_using_for tile_sizes [4,8] %target
  transform.yield %tiled, %for_loops#0, %marked_seq
}
... // do something with %tiled_target
transform.tune.alternatives<“par_or_seq_reprise”> {
  transform.param.cmpi eq %marker, %marked_par
  transform.loop.forall_to_parallel %loop
  transform.yield
}, {
  transform.param.cmpi eq %marker, %marked_seq
  transform.loop.coalesce %loop
  transform.yield
}
```

.mlir

lower /project

```
%marked_par = smt.int.constant 1
%marked_seq = smt.int.constant 0
%par_or_seq = smt.declare_const : !smt.int
...
smt.assert %par_or_seq_options
%par_or_seq_region0 = smt.eq %par_or_seq, %c0
%par_or_seq_region1 = smt.eq %par_or_seq, %c1
%marker = smt.declare_const : !smt.int
%marker_eq_marked_par = smt.eq %marker, %marked_par
%par_or_seq_region0_marking = smt.implies %par_or_seq_region0,
                                          %marker_eq_marked_par
smt.assert %region0_marking
%marker_eq_marked_seq = smt.eq %marker, %marked_seq
%par_or_seq_region1_marking = smt.implies %par_or_seq_region1,
                                          %marker_eq_marked_seq
smt.assert %par_or_seq_region1_marking

%par_or_seq_reprise = smt.declare_const : !smt.int
...
smt.assert %par_or_seq_reprise_options
%par_or_seq_reprise_region0 = smt.eq %par_or_seq_reprise, %c0
%par_or_seq_reprise_region1 = smt.eq %par_or_seq_reprise, %c1
%reprise_region0_cond = smt.implies %par_or_seq_reprise_region0,
                                    %marker_eq_marked_par
smt.assert %reprise_region0_cond
%reprise_region1_cond = smt.implies %par_or_seq_reprise_region1,
                                    %marker_eq_marked_seq
smt.assert %reprise_region1_cond
```

.mlir

convert

```
(declare-const par_or_seq Int)
(declare-const marker Int)
(declare-const par_or_seq_reprise Int)
...
(assert (implies (= par_or_seq 0) (= marker 1)))
(assert (implies (= par_or_seq 1) (= marker 0)))
(assert (implies (= par_or_seq_reprise 0) (= marker 1)))
(assert (implies (= par_or_seq_reprise 1) (= marker 0)))
```

.smtlib

Tied choices across sub-schedules

```
%marked_par = transform.param.constant 1
%marked_seq = transform.param.constant 0
%tiled_target, %loop, %marker = transform.tune.alternatives<“par_or_seq”> {
  %tiled, %forall_loop = transform.tile_using_forall tile_sizes [4,8] %target
  transform.yield %tiled, %forall_loop, %marked_par
}, {
  %tiled, %for_loops:2 = transform.tile_using_for tile_sizes [4,8] %target
  transform.yield %tiled, %for_loops#0, %marked_seq
}
... // do something with %tiled_target
transform.tune.alternatives<“par_or_seq_reprise”> {
  transform.param.cmpi eq %marker, %marked_par
  transform.loop.forall_to_parallel %loop
  transform.yield
}, {
  transform.param.cmpi eq %marker, %marked_seq
  transform.loop.coalesce %loop
  transform.yield
}
```
.mlir

lower /project &convert

```
(declare-const par_or_seq Int)
(declare-const marker Int)
(declare-const par_or_seq_reprise Int)
...
(assert (implies (= par_or_seq 0) (= marker 1)))
(assert (implies (= par_or_seq 1) (= marker 0)))
(assert (implies (= par_or_seq_reprise 0) (= marker 1)))
(assert (implies (= par_or_seq_reprise 1) (= marker 0)))
```
.smtlib

convert

SMT solver

```
(check-sat)
(get-assignment)
```
.smtlib

```
par_or_seq = 0
marker = 1
par_or_seq_reprise = 0
```

rewrite

```
%marked_par = transform.param.constant 1
%marked_seq = transform.param.constant 0
%tiled_target, %loop = transform.tile_using_forall tile_sizes [4,8] %target
... // do something with %tiled_target
transform.param.cmpi eq %marked_par, %marked_par
transform.loop.forall_to_parallel %loop
```
.mlir

Tied choices across sub-schedules

.mlir
```
%marked_par = transform.param.constant 1
%marked_seq = transform.param.constant 0
%tiled_target, %loop, %marker = transform.tune.alternatives<“par_or_seq”> {
  %tiled, %forall_loop = transform.tile_using_forall tile_sizes [4,8] %target
  transform.yield %tiled, %forall_loop, %marked_par
}, {
  %tiled, %for_loops:2 = transform.tile_using_for tile_sizes [4,8] %target
  transform.yield %tiled, %for_loops#0, %marked_seq
}
... // do something with %tiled_target
transform.tune.alternatives<“par_or_seq_reprise”> {
  transform.param.cmpi eq %marker, %marked_par
  transform.loop.forall_to_parallel %loop
  transform.yield
}, {
  transform.param.cmpi eq %marker, %marked_seq
  transform.loop.coalesce %loop
  transform.yield
}
```

lower /project &convert

.smtlib
```
(declare-const par_or_seq Int)
(declare-const marker Int)
(declare-const par_or_seq_reprise Int)
...
(assert (implies (= par_or_seq 0) (= marker 1)))
(assert (implies (= par_or_seq 1) (= marker 0)))
(assert (implies (= par_or_seq_reprise 0) (= marker 1)))
(assert (implies (= par_or_seq_reprise 1) (= marker 0)))
```

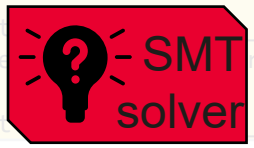
.mlir
```
%marked_par = transform.param.constant 1
%marked_seq = transform.param.constant 0
%tiled_target, %loop = transform.tile_using_forall tile_sizes [4,8] %target
... // do something with %tiled_target
transform.param.cmpi eq %marked_par, %marked_par
transform.loop.forall_to_parallel %loop
```

.mlir
```
%marked_par = transform.param.constant 1
%marked_seq = transform.param.constant 0
%tiled_target, %loop, %_ = transform.tile_using_for tile_sizes [4,8] %target
... // do something with %tiled_target
transform.param.cmpi eq %marked_seq, %marked_seq
transform.loop.coalesce %loop
```

rewrite

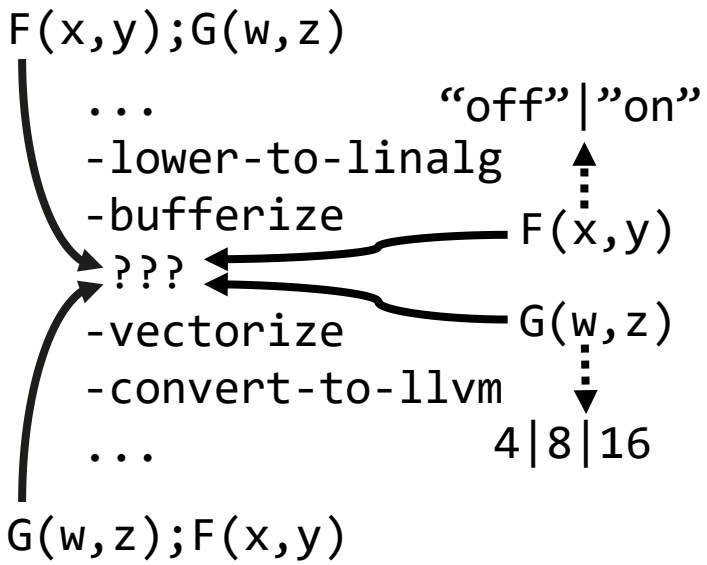
```
par_or_seq = 1
marker = 0
par_or_seq_reprise = 1
```

convert



.smtlib
```
(check-sat)
(get-assignment)
```

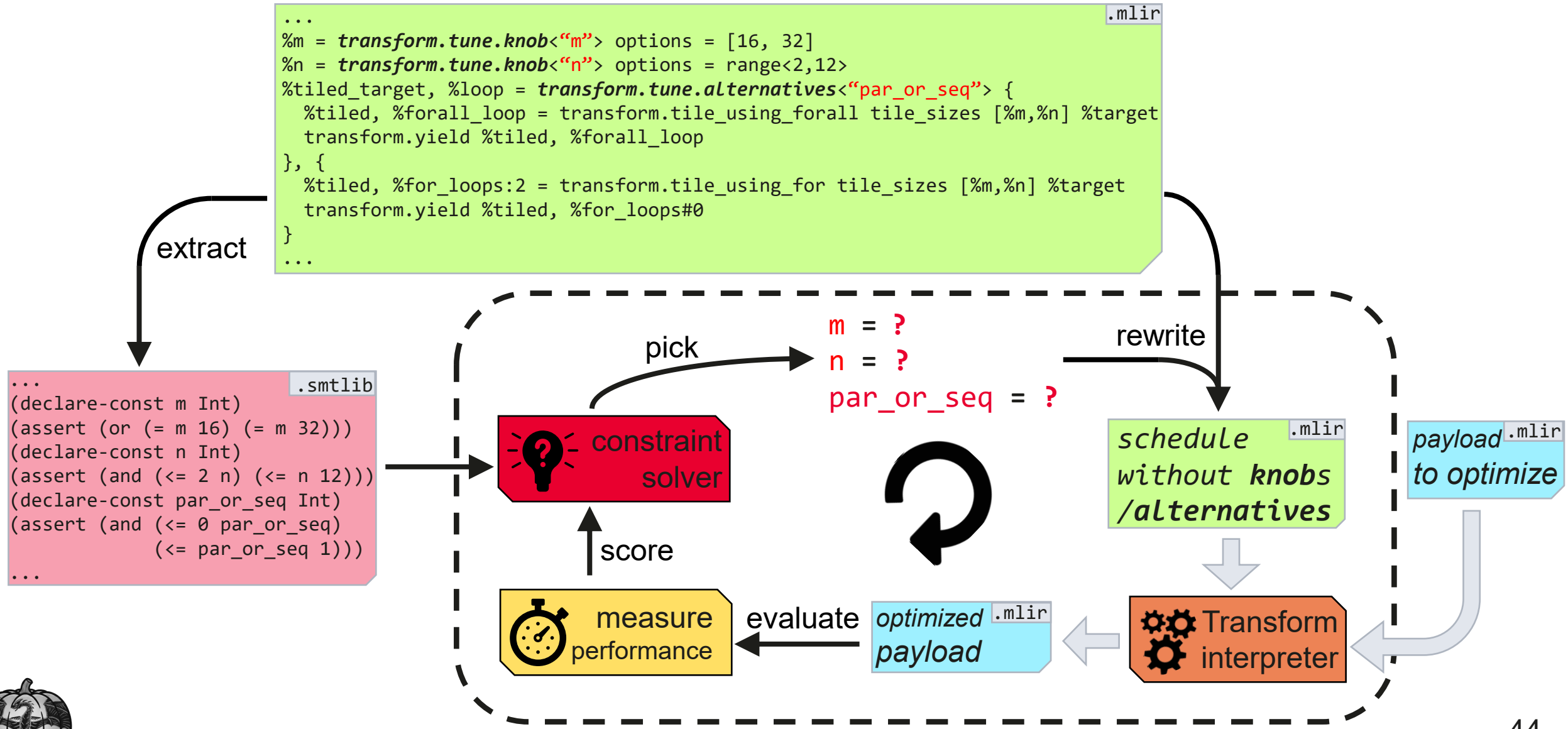
An average plan for an average day … in IR



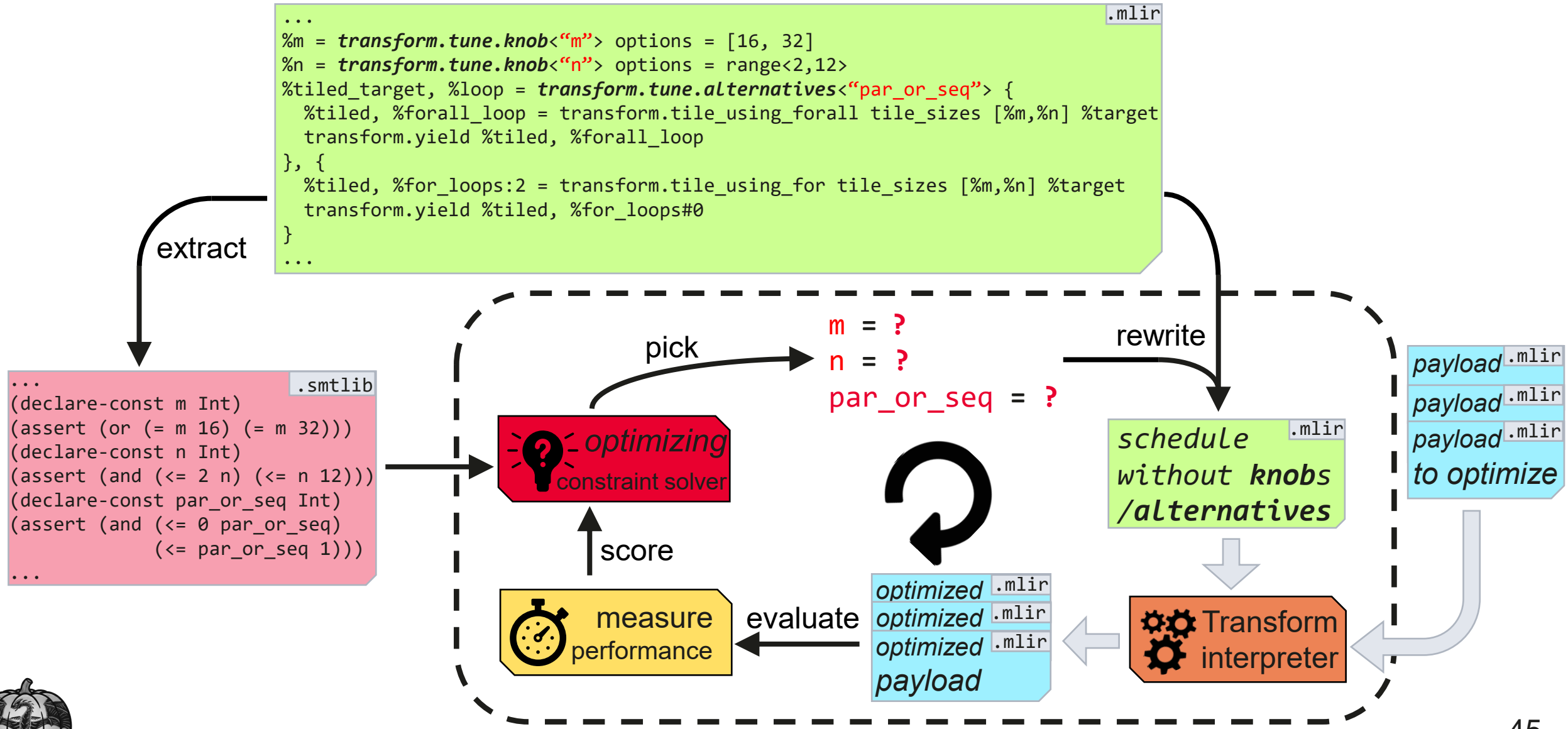
.mlir

```
...
%tensorized = transform.apply_registered_pass “lower-to-linalg” to %target
%bufferized = transform.apply_registered_pass “bufferize” to %tensorized
%x = transform.tune.knob<“w”> options = [4,8,16]
%y = transform.tune.knob<“z”> options = ...
%optimized = transform.tune.alternatives<“F_or_no_F”> {
  %pre, %G_performed = transform.tune.alternatives<“G_before_F_or_not”> {
    %Ged = transform.apply_registered_pass “G” with options { “w” = %w, “z” = %z } to %bufferized
    transform.yield %Ged, %c1
  }, {
    transform.yield %bufferized, %c0
  }
  %a = transform.tune.knob<“x”> options = [“off”,”on”]
  %b = transform.tune.knob<“y”> options = ...
  %Fed = transform.apply_registered_pass “F” with options { “x” = %x, “y” = %y } to %bufferized
  %res = transform.tune.alternatives<“G_after_F_or_not”> {
    transform.param.cmpi eq %G_performed, %c0 // crash in case G got performed before F
    %Ged = transform.apply_registered_pass “G” with options { “w” = %w, “z” = %z } to %Fed
    transform.yield %Ged
  }, {
    transform.yield %Fed
  }
  transform.yield %res
}, {  // Only G, no F
 %Ged = transform.apply_registered_pass “G” with options { “w” = %w, “z” = %z } to %bufferized
 transform.yield %Ged
}
%vectorized = transform.apply_registered_pass “vectorize” to %optimized
%llvmed = transform.apply_registered_pass “convert-to-llvm” to %vectorized
...
```

Mechanized optimization of schedules



Mechanized optimization of schedules



In summary

Choices for parameters on transforms can be “reified” into IR

- This gives knob ops and non-deterministic reading of schedules

Choice among transform sequences can likewise be reified

- This goes beyond (parameter) auto-tuning, without the intractability of auto-scheduling

Joint constraints allow for precise control over search space

- So that engineers can specify which schedules they are willing to consider

Search for find constraint-satisfying assignments can be handed off, e.g., to a SMT solver

- Whereupon the assignments can be used to rewrite to executable schedules

Questions?