Type-Directed Automatic Incrementalization
Yan Chen, Joshua Dunfield, and Umut A. Acar (Max Planck Institute for Software Systems)

Self-Adjusting Computation
- Dynamic changes are pervasive: compilation, physical simulation, iterative algorithms, GUIs, …
- Self-adjusting computation provides language support to *incrementalize* computations, so they respond to dynamic changes efficiently.
- Traditional explicit approaches to self-adjusting computation require substantial programmer effort.

Implicit Self-Adjusting Computation
- To scale up to large programs, we allow *implicitly* self-adjusting programs (Chen et al., ICFP 2011), where changeable data is marked in *type annotations*:

```sml
fun multiply (A, B) =
  let val Tb = transpose B
  in
  map (A, fn row ⇒
    map (Tb, fn col ⇒
      reduce (map2 (row, col, fn (a,b) ⇒ a*b)), 0,)
    fn (x, res) ⇒ x + res))
  end
```

Explicit Self-Adjusting (SAC) program
- Modifiable: a traceable reference type
- Read: reads from a modifiable; if the modifiable changes, runtime will reexecute the read body
- Write: writes to a modifiable
- Mod: creates a modifiable, enters changeable mode; changeable mode must end with a write or a function call

Implementation
- Our modified MLton: modified phases, new phases, unmodified phases
- **Front end** propagates annotations ($C$) to maintain necessary invariants of explicit SAC.
- **Translate** uses type information (int $C$, etc.) to automatically insert Mod, Read, and Write operations on changeable data.
- **Optimize** eliminates Read-Write-Mod and other redundant operations.
- The resulting explicit self-adjusting program can be compiled normally.

Translation preserves types (under a translation from $C$-types to mod-types) and meaning (compared to a conventional program).

Results
- **Mergesort**
- **Blocked Matrix Multiply**
- **Ray Tracer**

Comparison to Previous Work