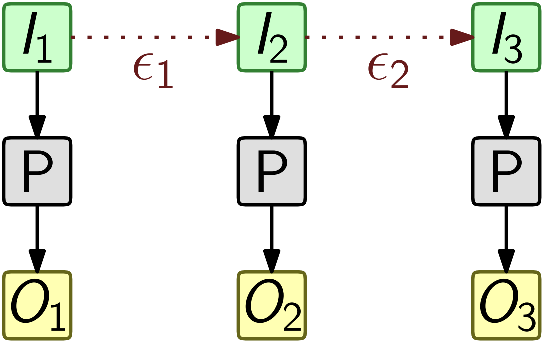
CEAL: A C-based Language for Self-Adjusting Computation

Matthew Hammer   Umut Acar   Yan Chen

Toyota Technological Institute at Chicago

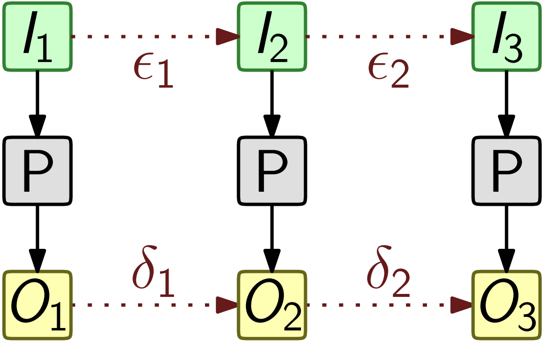
PLDI 2009

Self-adjusting computation



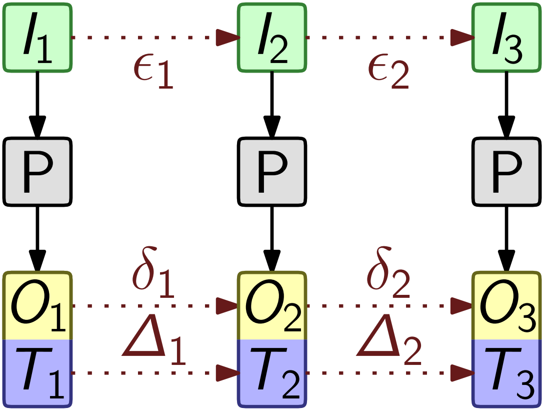
- Programs usually run **from-scratch** on new inputs

Self-adjusting computation



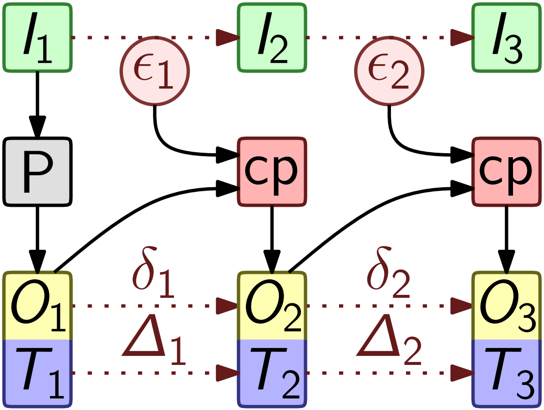
- Programs usually run **from-scratch** on new inputs
- Interested when small input change $\Rightarrow$ small output change

Self-adjusting computation



- Include **program trace** T with program output
- **Idea**: Small ϵ and small δ often implies small Δ

Self-adjusting computation



- **Initial run** records a program trace
- **Change propagation** (**cp**) updates the output & trace

Goal of self-adjusting computation

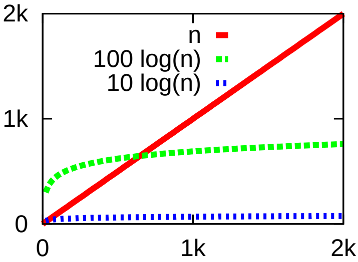
Given input change (ϵ), update output & trace in time proportional to trace distance (Δ)

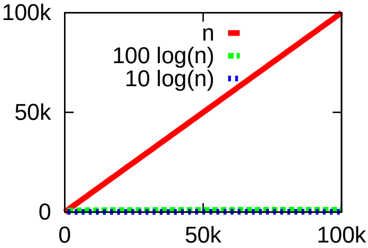
Application	From-scratch	Insertion/deletion
List Primitives	$O(n)$	$O(1)$
Quicksort	$O(n \cdot \log n)$	$O(\log n)$
Mergesort	$O(n \cdot \log n)$	$O(\log n)$
Convex Hulls (2d)	$O(n \cdot \log n)$	$O(\log n)$
Convex Hulls (3d)	$O(n \cdot \log n)$	$O(\log n)$

From-scratch time vs trace distance for an insertion/deletion

Linear vs logarithmic time

- Interesting problems take $O(n)$ time (or more)
- Suppose we can update output in $O(\log n)$ time
- Is the speedup worth it?





$O(n)$ increases exponentially faster than $O(\log n)$

Challenges for Self-Adjusting Computation

Challenge 1

- Impractical to trace every operation
- **What operations should be traced?**

Challenge 2

- Trace must support efficient incremental updates
- **How should the trace be structured?**

What operations should be traced?

Idea: Distinguish between **stable** and **changeable** data

- Programmer manages changeable data in **modrefs** (modifiable references)
- Analogous to conventional references
- Trace records modref operations

Modref operations

`modref_t*` **modref**`()`	Create an empty modref
`void` **write**`(modref_t *m, void *p)`	Write to a modref
`void*` **read**`(modref_t *m)`	Read from a modref

Example: Evaluating expression trees

```
ceal eval (modref_t *in, modref_t *out) {
  node_t *node = read (in);
  if (node->kind == LEAF)
    write (out, node->leaf_value);
  else {
    modref_t *m_a = modref ();
    modref_t *m_b = modref ();
    eval (node->left_child, m_a);
    eval (node->right_child, m_b);
    int a = read (m_a);
    int b = read (m_b);

    if (node->binary_op == PLUS) {
      write (out, a + b);
    } else {
      write (out, a - b);
    }
  }
}
```

Key Idea

Input & output stored in **modrefs**

TODO: illustrate execution of the two cases

How is trace structured? how do we update it?

Need to identify & record dependencies between data & code

Idea: When a modref is changed, rerun code with new value

How is trace structured? how do we update it?

Need to identify & record dependencies between data & code

Idea: When a modref is changed, rerun code with new value

What code do we rerun?

How is trace structured? how do we update it?

Need to identify & record dependencies between data & code

Idea: When a modref is changed, rerun code with new value
What code do we rerun?

Normal form

Every read followed by a **tail call**

$$x \texttt{ = read(}m\texttt{); tail } f\texttt{(}x\texttt{, }y\texttt{)}$$

- Use of m's value recorded as a closure of f
- Closure can be rerun if & when m changes

Tracing return values

TODO: make this slide more concise

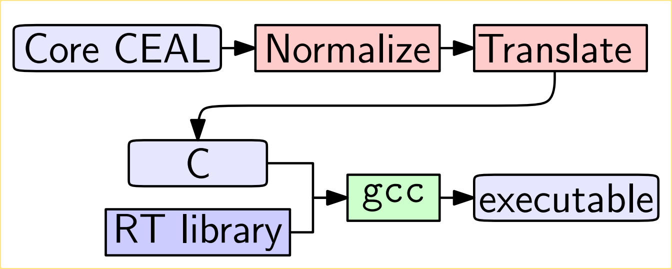
- How to trace & rerun functions with return values?
- How do we rerun the caller without recording call stack?

Idea: Return results through **modref** arguments

- Don't want to record call stack
- Restrict all functions with **reads** to return **void**
- **Destination-passing style** returns results in modrefs
- $\Rightarrow$ **modref**s track all callee-to-caller dataflow

Compilation Overview

Goal: Compile CEAL into C code.
Target C code is linked with a runtime library.



CEAL to C: a two step process

- **Normalize** CEAL code (put into **normal form**)
- **Translate** the (normal form) CEAL code to C

Normalization

Normalization via control-flow graphs

Idea: Transform the program as a control-flow graph

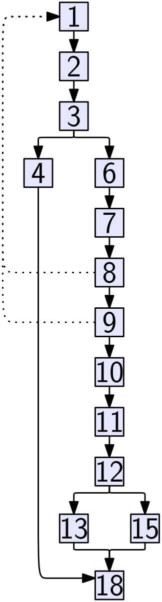
Nodes
- a designated **root node**
- a **function node**
- a command (read, write, etc.), conditional, or return statement

Entry Nodes
- a **function entry** $\equiv$ a function node
- a **read entry** $\equiv$ successor of a read

Edges
- Control edges: (tail) call & goto
- **Entry edges**: from root to entry nodes

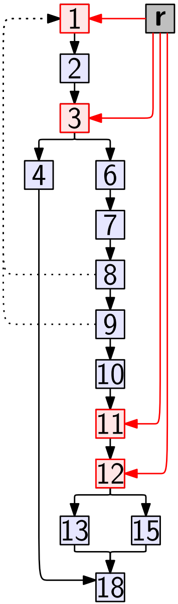
Example: Program graph for eval

```
ceal eval (modref_t *in, modref_t *out) {
  node_t *node = read (in);
  if (node->kind == LEAF) {
    write (out, node->leaf_value);
  } else {
    modref_t *m_a = modref ();
    modref_t *m_b = modref ();
    eval (node->left_child, m_a);
    eval (node->right_child, m_b);
    int a = read (m_a);
    int b = read (m_b);
    if (node->binary_op == PLUS) {
      write (out, a + b);
    } else {
      write (out, a - b);
    }
  }
  return; }
```



Example: Program graph for eval

```
ceal eval (modref_t *in, modref_t *out) {
  node_t *node = read (in);
  if (node->kind == LEAF) {
    write (out, node->leaf_value);
  } else {
    modref_t *m_a = modref ();
    modref_t *m_b = modref ();
    eval (node->left_child, m_a);
    eval (node->right_child, m_b);
    int a = read (m_a);
    int b = read (m_b);
    if (node->binary_op == PLUS) {
      write (out, a + b);
    } else {
      write (out, a - b);
    }
  }
return; }
```



Dominator Relation, Dominator Trees

Def: Dominator relation

Node a **dominates** b if every path from root to b contains a

Def: Immediate dominator relation

Node a is the **immediate dominator** of b if

- $a \neq b$
- a dominates b
- Every other dominator of b dominates a

Every node has a (unique) immediate dominator, except **root**.

Def: Dominator tree

Immediate dominator relation forms a tree (root is root node)

Dominators & critical nodes

Dominator examples

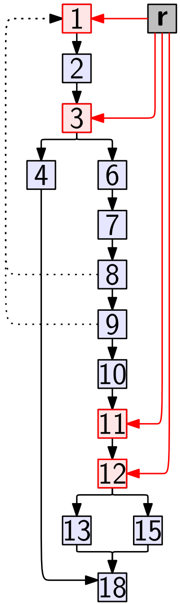
- Root **r** dominates all nodes
- 1 dominates 2, but not 3
- 3 dominates 4 & 6–10, but not 11
- 12 dominates 13 & 15, but not 18

Root **r** is **immediate dominator** of

- Every entry node
- Nodes not dominated by any entry (18)

Define: Critical nodes

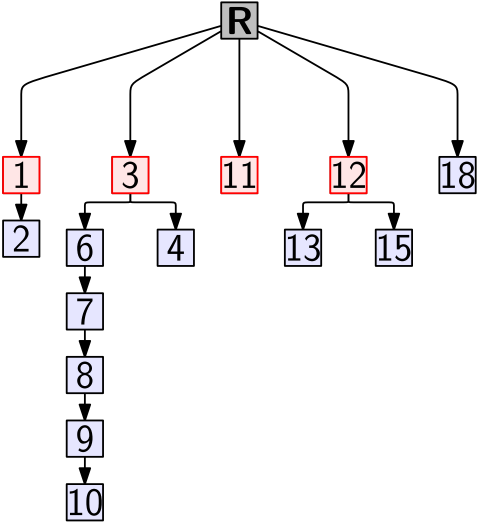
Nodes immediately dominated by the root



Units & cross-unit edges

- Define **critical nodes** as root's children:

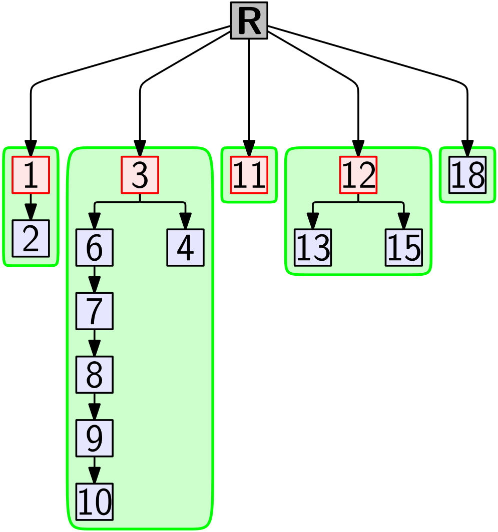
  Nodes 1, 2, 11, 12 & 18.



Units & cross-unit edges

- Define **critical nodes** as root's children:

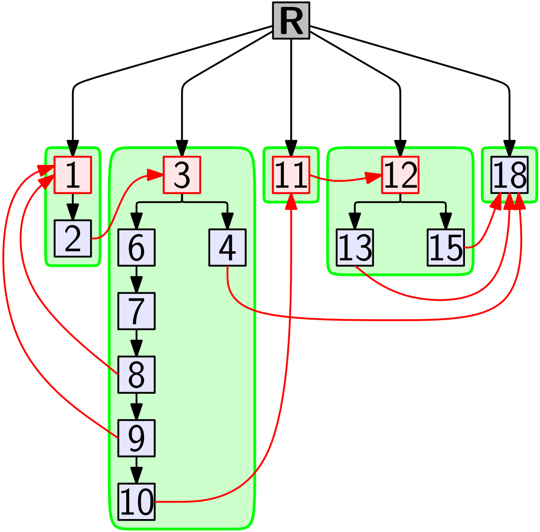
  Nodes 1, 2, 11, 12 & 18.
- Define **units** as subtrees of critical nodes



Units & cross-unit edges

- Define **critical nodes** as root's children:

  Nodes 1, 2, 11, 12 & 18.
- Define **units** as subtrees of critical nodes
- **Lemma**: every **cross-unit** edge targets a critical node.
- **Corrollary**: If each unit becomes a separate function, then cross-unit edges can become calls.



Normalization: The Algorithm

Main Ideas:

- Units $\rightsquigarrow$ separate functions
- Cross-unit edges $\rightsquigarrow$ tail calls (args $\equiv$ live vars)

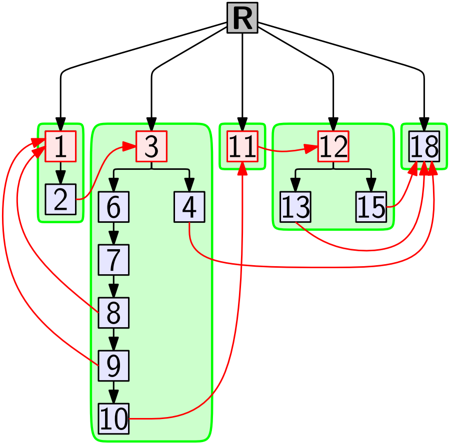
Algorithm

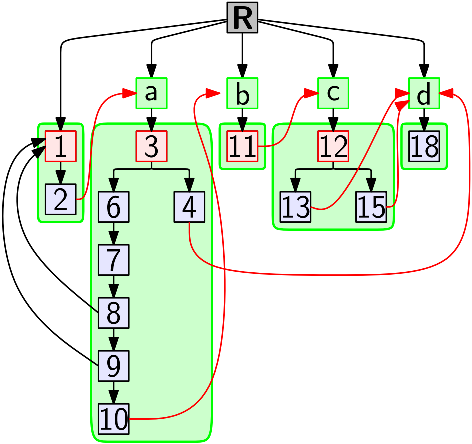
1. Compute the dominator tree
2. For each critical node, not yet a function node:
   - Create a new function node for unit
   - Redirect incoming critical edges to new function node (not always necessary; omitting minor details)

Example: New functions, Redirected edges

Before redirection

After redirection





Node 1 already a function node, so no new function needed

Example: Output graph as output code

```
1   ceal eval (modref_t *in, modref_t *out) {
2     node_t *node = read (in); tail eval_a (node, out);
    }

a   ceal eval_a (node_t *node, modref_t *out) {
3     if (node->kind == LEAF) {
4       write (out, node->leaf_value); tail eval_d ();
5     } else {
6-9     ...
10      int a = read (m_a); tail eval_b (out, a, m_b);
17    }
    }

b   ceal eval_b (modref_t *out, int a, modref_t *m_b) {
11    int b = read (m_b); tail eval_c (out, a, b);
    }

c   ceal eval_c (modref_t *out, int a, int b) {
12    if (node->binary_op == PLUS) {
13      write (out, a + b); tail eval_d ();
14    } else {
15      write (out, a - b); tail eval_d ();
16    }
    }

d   ceal eval_d () {
18    return;
    }
```

Normal Form

- Original function split into five
- Cross-unit edges become tail calls
- Tail call follows each **read**

Translation

Translation Overview

Translation Basics

- Translation introduces closures for tail calls
- For **reads**: associates closure with read modref
- Uses a **trampoline** to run closures iteratively

Translation Overview

Translation Basics

- Translation introduces closures for tail calls
- For **reads**: associates closure with read modref
- Uses a **trampoline** to run closures iteratively

Selective trampolining

- Only need to record closures for **reads**
- So, only trampoline tail calls that follow **reads**
- Other "tail calls" treated like ordinary calls
- Stack grows only temporarily (until a **read**)

Translation Overview

Translation Basics

- Translation introduces closures for tail calls
- For **reads**: associates closure with read modref
- Uses a **trampoline** to run closures iteratively

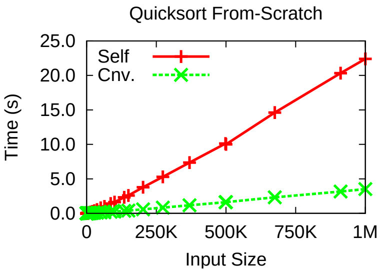
Selective trampolining

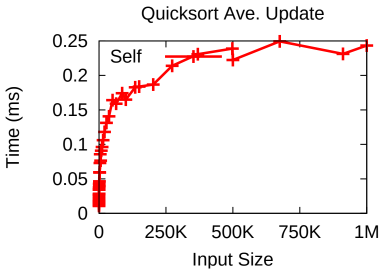
- Only need to record closures for **reads**
- So, only trampoline tail calls that follow **reads**
- Other "tail calls" treated like ordinary calls
- Stack grows only temporarily (until a **read**)

See paper for details

Performance Evaluation

Evaluation Example: Quicksort





Overhead: about 6x (a constant)

Speedup: 1.4×10^4 (increases linearly with input size)

Results summary

		From-Scratch			Propagation	
Application	n	Cnv.	Self.	O.H.	Ave. Update	Speedup
`filter`	10.0M	0.5	7.4	14.2	2.1×10^{-6}	2.4×10^{5}
`map`	10.0M	0.7	11.9	17.2	1.6×10^{-6}	4.2×10^{5}
`reverse`	10.0M	0.6	11.9	18.8	1.6×10^{-6}	3.9×10^{5}
`minimum`	10.0M	0.8	10.9	13.8	4.8×10^{-6}	1.6×10^{5}
`sum`	10.0M	0.8	10.9	13.9	7.0×10^{-5}	1.1×10^{4}
`quicksort`	1.0M	3.5	22.4	6.4	2.4×10^{-4}	1.4×10^{4}
`quickhull`	1.0M	1.1	12.3	11.5	2.3×10^{-4}	4.6×10^{3}
`diameter`	1.0M	1.0	12.1	12.0	1.2×10^{-4}	8.3×10^{3}
`exptrees`	10.0M	1.0	7.2	7.2	1.4×10^{-6}	7.1×10^{5}
`mergesort`	1.0M	6.1	37.6	6.1	1.2×10^{-4}	5.1×10^{4}
`distance`	1.0M	1.0	11.0	11.0	1.3×10^{-3}	7.5×10^{2}
`tcon`	1.0M	2.6	20.6	7.9	1.0×10^{-4}	2.5×10^{4}

Average Overhead 6–19x

Average Speedups 3.6×10^4 (for $n = 1$M)
1.4×10^5 (for $n = 10$M)

Related Work

Other self-adjusting/incremental language support:

Acar et. al., PLDI'05	SAC library for ML (Sting)
Shankar & Bodík, PLDI'07	Invariant checks in Java (Ditto)
Hammer & Acar, ISMM'08	SAC library for C
Ley-Wild et. al., ICFP'08	DeltaML language & compiler

DeltaML is most comparable system

- Compiler support for general-purpose SAC
- Similar modref-like primitives
- Similar benchmarks

CEAL *vs* DeltaML: Summary

Normalized Measurements (DeltaML / CEAL)

App.	From-Scratch	Ave. Update	Max Live
`filter`	9.3	6.2	4.4
`map`	9.3	7.1	4.4
`reverse`	8.0	5.8	4.2
`minimum`	4.6	8.8	2.9
`sum`	4.6	3.5	2.9
`quicksort`	26.9	15.6	4.8
`quickhull`	5.1	3.3	1.1
`diameter`	5.8	4.3	1.5

- **From-Scratch** CEAL 5–27 times faster (9 on average)
- **Change propagation** CEAL 3–9 times faster (7 on average)
- **Max live** CEAL uses up to 5 times less space (3 on average)

Concluding remarks

CEAL: In Summary

- C-based language for self-adjusting computation
- Compiles directly to (portable) C code
- Promising performance results

On-going & future directions

- Support for return values
- Implicit modifiable operations (using type annotations)
- Finer-grained code dependencies for reads
  (At what point can re-execution stop?)

Thank You!

Questions?