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

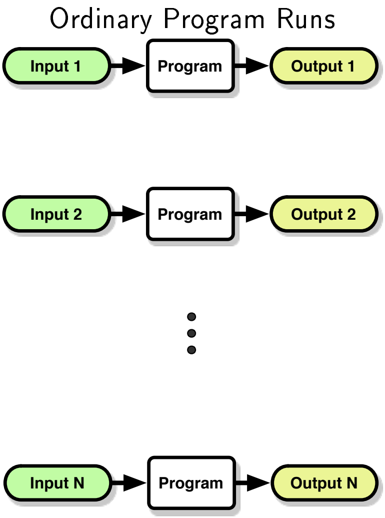
Matthew Hammer Umut Acar Yan Chen

Toyota Technological Institute at Chicago

February 13, 2009

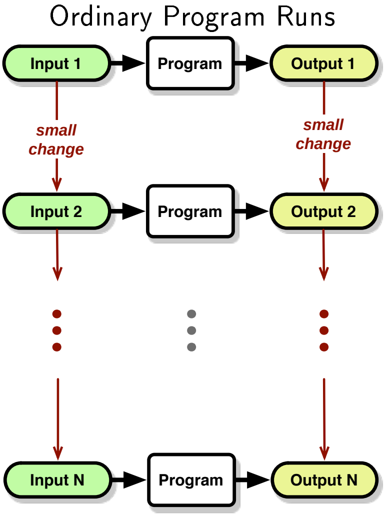
Overview of Talk

- Background (self-adjusting computation)
- CEAL via example
- Compilation process: CEAL to C
- Performance results & comparison to SaSML
- Conclusion



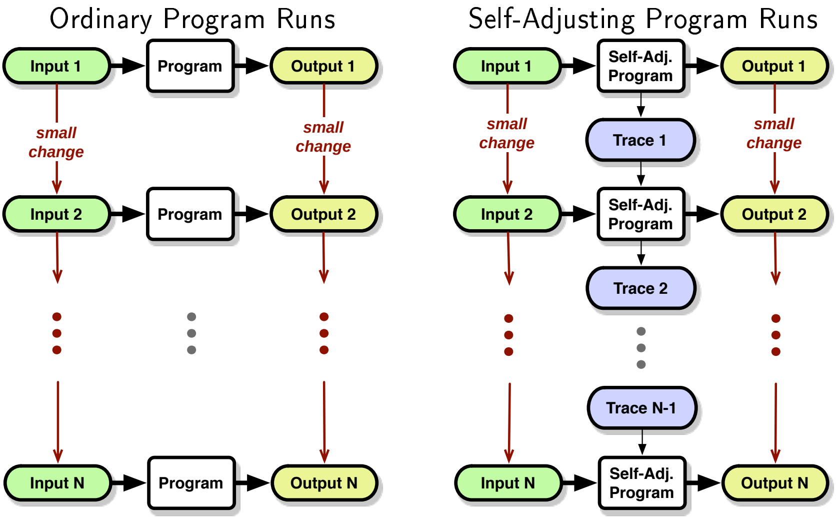
- Ordinary programs often run repeatedly on changing input.
- What if input and output change by only small increments?

Self-Adjusting Computation



- Ordinary programs often run repeatedly on changing input.
- What if input and output change by only small increments?

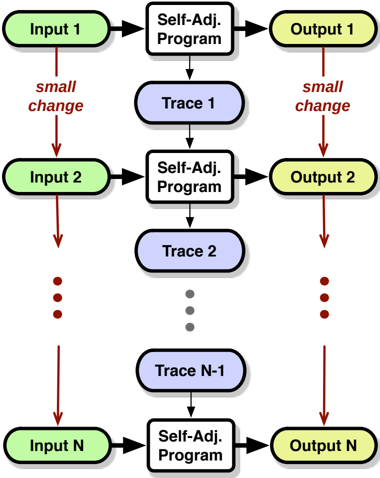
Self-Adjusting Computation



Self-Adjusting Computation

- Record execution in a program trace
- When input changes, a **change propagation** algorithm updates the output and trace as if the program was run "from-scratch".
- Tries to reuse past computation when possible

Self-Adjusting Program Runs



Motivation : Incremental change is pervasive.

Many applications encounter data that changes slowly or *incrementally* over time.

- Applications that interact with a physical environment.
  E.g., Robots.
- Applications that interact with a user.
  E.g., Games, Editors, Compilers, etc.
- Applications that rely on modeling or simulation.
  E.g., Scientific Computing, Computational Biology, Motion Simulation.

Self-Adjusting Computation

Previous work has shown effectiveness for many applications:

List primitives (map, reverse, . . .)	$O(1)$	
Sorting: mergesort, quicksort	$O(\log n)$	
2D Convex hulls	$O(\log n)$	[ESA '06]
Tree contraction [Miller, Reif '85]	$O(\log n)$	[SODA '04].
3D Convex Hulls	$O(\log n)$	[SCG '07]
Meshing in 2D and 3D	$O(\log n)$	[FWCG '07]
Bayesian Inference on Trees	$O(\log n)$	[NIPS '07]
Bayesian Inference on Graphs	$O(s^d \log n)$	[UAI '08]

All bounds are randomized (expected time) and are within an expected constant factor of optimal or best known-bounds.

Key programming concepts

modifiable references (**modref** for short)

- analogous to ordinary mutable memory
- each hold one word (e.g., a pointer)
- primitives to create/read/write them

mutator & core programs: two stratified levels

1. mutator runs core, whose execution is traced
2. mutator inspects core output
3. mutator typically loops:
   - 3.1 mutator changes core input
   - 3.2 mutator invokes automatic **change propagation** mechanism
   - 3.3 mutator inspects (automatically updated) core output

CEAL primitives

Core language

`modref_t* modref()` Create empty modrefs

`void write(modref_t *m, void *p)` Write to modrefs

`void* read(modref_t *m)` Read from modrefs

`void* alloc(int sz, f, ...)` Allocate and initialize

Meta (aka mutator) language

`void run_core(f, ...)` Run core program

`void* deref(modref_t* m)` Inspect modrefs

`void modify(modref_t* m, void *p)` Modify modrefs

`void propagate()` Propagate modifications

Compilation Overview

- INPUT: CEAL program (set of core and mutator functions)
- CEAL primitives provided by ordinary header file
- Our run-time library provides tracing, change propagation & memory management.
- CIL [Necula et al] parses/type-checks CEAL as C code
- We extend CIL to:
  - Build our own CFG representation
  - ⋆ Analyze & transform CFG according to reads (**normalization**)
  - ⋆ Translate CFG into C with runtime calls
- last, use GCC to compile and link target program

Modifiable lists

```
typedef struct {
  int hd;
  modref_t *tl;
} cons_t;

void cons_init(cons_t *c, int h) {
  c->hd = h;
  c->tl = modref();
}
```

```
Cons(h) ≡ alloc(sizeof(cons_t), cons_init, h)
```

- Cells allocated with given head, empty (modifiable) tail
- Cells are reused based on matching heads (and re-evaluation context)

Example: Merging two sorted lists

```
ceal merge(L1, L2, D) {
  c1 = read(L1);
  c2 = read(L2);
  while(c1 && c2) {
    if(c1->hd < c2->hd) {
      c' = Cons(c1->hd);
      c1 = read(c1->tl);
    } else {
      c' = Cons(c2->hd);
      c2 = read(c2->tl);
    }
    write(D, c');
    D = c'->tl;
  }
  if(c1)
    write(D, c1);
  else
    write(D, c2);
}
```

- L_1, L_2, D : `modref_t*`
- c_1, c_2, c' : `cons_t*`

Example (from-scratch) input/output

Pre:

$L_1 = [1, 2, 5]$, $L_2 = [3, 4, 6]$

Post:

$D = [1, 2, 3, 4, 5, 6]$

Tracing, re-evaluation and memo-matching

Main Ideas

- Each read has a *"context"* using the read value
- Contexts correspond to a *closure* (or *thunk* or *continuation*, etc.) of some kind; we include it in trace.
- When read value changes, re-evaluate context
- When current read-context matches past one, reuse it

Questions

- How much context do we have to capture in the trace?
  - Do we include the call stack?
  - No, assume no return values (dest-passing style)
- How do we represent it for C programs in a C runtime?
  - Want: a function pointer and args (a closure)
  - Goal: make read-contexts correspond to invocations

Example: Merging two sorted lists

```
ceal merge(L1, L2, D) {
  c1 = read(L1);
  c2 = read(L2);
  while(c1 && c2) {
    if(c1->hd < c2->hd) {
      c' = Cons(c1->hd);
      c1 = read(c1->tl);
    } else {
      c' = Cons(c2->hd);
      c2 = read(c2->tl);
    }
    write(D, c');
    D = c'->tl;
  }
  if(c1)
    write(D, c1);
  else
    write(D, c2);
}
```

Example (from-scratch) input/output

Pre:

$L_1 = [1, 2, 5]$, $L_2 = [3, 4, 6]$

Post:

$D = [1, 2, 3, 4, 5, 6]$

Change Propagation

- Assume L_1 and L_2 "alternate" sufficiently (e.g., (2,3), (4,5), (5,6))
- Update for insertion/deletion in $O(1)$:
  - Wake-up and re-evaluate until we redo previous alternation
  - Implies c_1, c_2, D and c' match previous a read-state
  - Memo-match rest of computation

CEAL Core Language / Control-Flow Graphs (CFGs)

$$
\begin{array}{llcl}
\textit{Values} & v & ::= & \ell \mid n \\
\textit{Expressions} & e & ::= & n \mid x \mid \oplus(\overline{x}) \mid x[y] \\
\textit{Commands} & c & ::= & x := e \\
 & & \mid & x := \texttt{modref}() \\
 & & \mid & x := \texttt{read}\ y \\
 & & \mid & \texttt{write}\ x := y \\
 & & \mid & x := \texttt{alloc}\ y\ f\ \overline{z} \\
 & & \mid & \texttt{call}\ f\ (\overline{x}) \\
\textit{Jumps} & j & ::= & \texttt{goto}\ l \\
 & & \mid & \texttt{tail}\ f\ (\overline{x}) \\
\textit{Basic Blocks} & b & ::= & \{l : \texttt{done}\} \\
 & & \mid & \{l : \texttt{cond}\ x\ j_1\ j_2\} \\
 & & \mid & \{l : c\ ;\ j\} \\
\textit{Fun. Defs} & F & ::= & f\ (\overline{x})\ \{\overline{y}; \overline{b}\} \\
\textit{Programs} & P & ::= & \overline{F}
\end{array}
$$

Normal form programs

Define: **Normal Form**

Read blocks have the form: $\{l : x = \texttt{read}\ y;\ j\}$

Not Normal: $\{l : x = \texttt{read}\ y;\ \texttt{goto}\ l'\}$

Normal: $\{l : x = \texttt{read}\ y;\ \texttt{tail}\ f(x, \ldots)\}$

Program is in **Normal Form** iff each read block is in NF.

Normalization

Goal: put all read blocks into normal form.

- Repartition basic blocks (as coarsely as possible)
- Turn successors of read-blocks into function entry nodes.
- Non-local gotos become `tail` jumps
- Preserve the program's:
  - Size (e.g., total basic blocks)
  - Input/output behavior
  - Time/space requirements

Dominators

Assume: CFG has designated **root** node,
every node is reachable from root.

Def: Dominator relation

a **dominates** b if every path from root to b contains a.
(note: every node dominates itself).

Def: Immediate dominator relation

a is the (unique) **immediate dominator** of b if

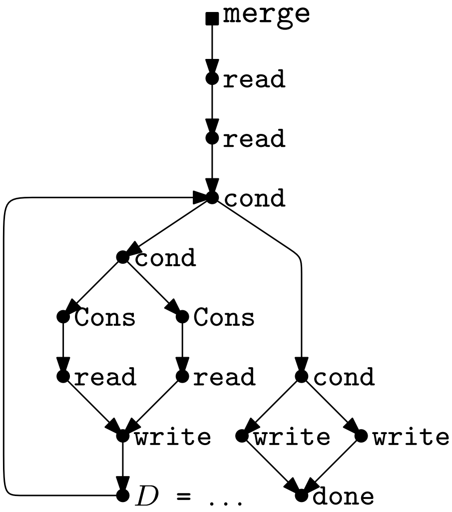
- $a \neq b$
- a dominates b
- c dominates b implies c dominates a

Def: Dominator tree

Immediate dominator relation forms a tree, rooted at the CFG root.

Example: Normalizing **merge**

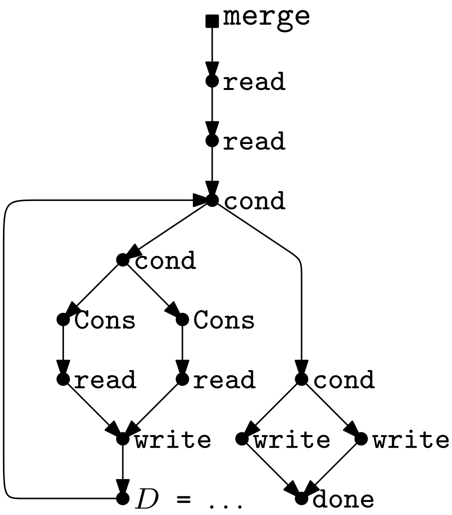
```
ceal merge(L1, L2, D) {
  c1 = read(L1);
  c2 = read(L2);
  while(c1 && c2) {
    if(c1->hd < c2->hd) {
      c' = Cons(c1->hd);
      c1 = read(c1->tl);
    } else {
      c' = Cons(c2->hd);
      c2 = read(c2->tl);
    }
    write(D, c');
    D = c'->tl;
  }
  if(c1)
    write(D, c1);
  else
    write(D, c2);
}
```



Example: Normalizing **merge**

Normalization

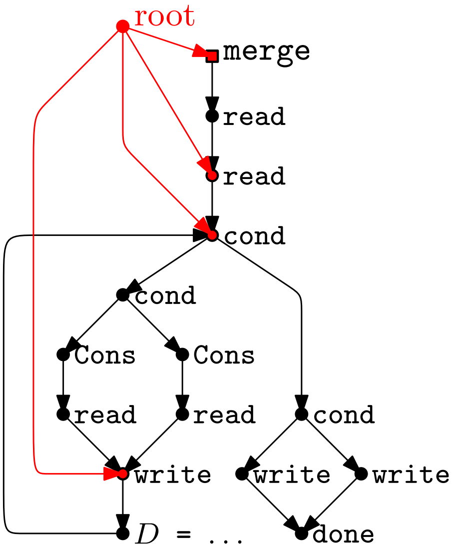
- Compute CFG
- Add **root node** & edges to **entry nodes**
- Compute dominator tree
- Root subtrees are **units**
- CFG partitioned by **units**, Each **unit** is a new CFG
- Cross-unit **gotos** become **tail jumps** to new funs, Live variables become arguments.



Example: Normalizing **merge**

Normalization

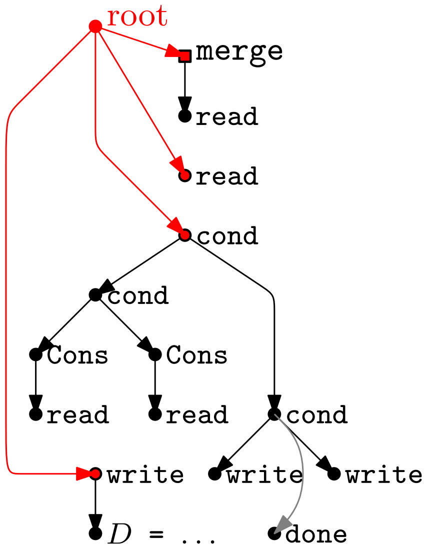
- Compute CFG
- Add **root node** & edges to **entry nodes**
- Compute dominator tree
- Root subtrees are **units**
- CFG partitioned by **units**, Each **unit** is a new CFG
- Cross-unit **gotos** become **tail jumps** to new funs, Live variables become arguments.



Example: Normalizing **merge**

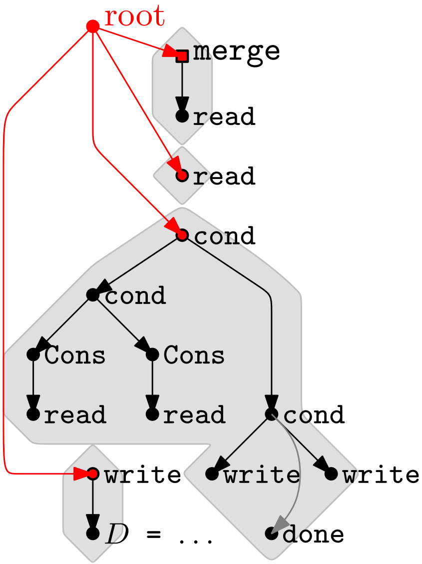
Normalization

- Compute CFG
- Add **root node** & edges to **entry nodes**
- Compute dominator tree
- Root subtrees are **units**
- CFG partitioned by **units**, Each **unit** is a new CFG
- Cross-unit **gotos** become **tail jumps** to new funs, Live variables become arguments.



Normalization

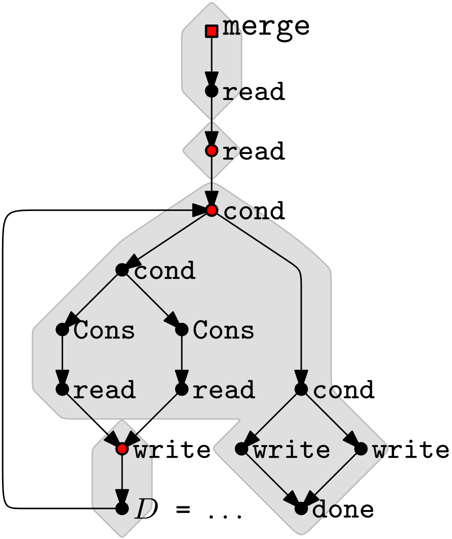
- Compute CFG
- Add **root node** & edges to **entry nodes**
- Compute dominator tree
- Root subtrees are **units**
- CFG partitioned by **units**, Each **unit** is a new CFG
- Cross-unit **gotos** become **tail jumps** to new funs, Live variables become arguments.



Example: Normalizing **merge**

Normalization

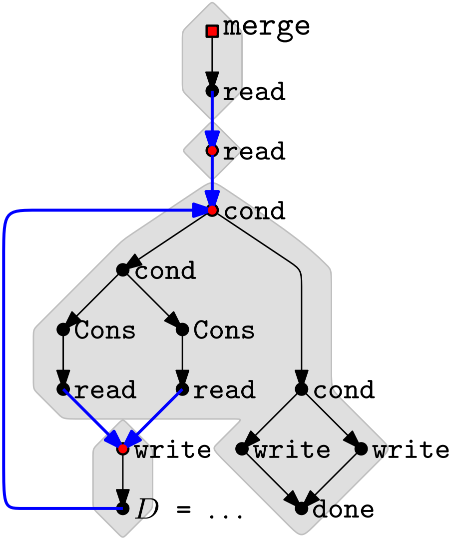
- Compute CFG
- Add **root node** & edges to **entry nodes**
- Compute dominator tree
- Root subtrees are **units**
- CFG partitioned by **units**, Each **unit** is a new CFG
- Cross-unit **gotos** become **tail jumps** to new funs, Live variables become arguments.



Example: Normalizing **merge**

Normalization

- Compute CFG
- Add **root node**
  & edges to **entry nodes**
- Compute dominator tree
- Root subtrees are **units**
- CFG partitioned by **units**,
  Each **unit** is a new CFG
- Cross-unit **gotos** become
  tail jumps to new funs,
  Live variables become
  arguments.



Example: Normalized **merge**

```
ceal merge(L₁, L₂, D) {
  c₁ = read(L₁);
  tail merge₂(c₁, L₂, D);
}

ceal merge₂(c₁, L₂, D) {
  c₂ = read(L₂);
  tail merge₃(c₁, c₂, D);
}

ceal merge₄(c₁, c₂, c′, D) {
  write(D, c′);
  D = c′->tl;
  tail merge₃(c₁, c₂, D);
}
```

```
ceal merge₃(c₁, c₂, D) {
  if(c₁ && c₂) {
    if(c₁->hd < c₂->hd) {
      c′ = Cons(c₁->hd);
      c₁ = read(c₁->tl);
      tail merge₄(c₁, c₂, c′, D);
    } else {
      c′ = Cons(c₂->hd);
      c₂ = read(c₂->tl);
      tail merge₄(c₁, c₂, c′, D);
    }
  } else {
    if(c₁)
      write(D, c₁);
    else
      write(D, c₂);
  }
}
```

Normalization Properties

Basic Properties

- Works on arbitrary control-flow (e.g., non-natural loops)
- Preserves program semantics, space & time
- Can be implemented to run in linear-time,
  (Assuming we use linear-time dominator tree algo)

Per-function transformation

Normalization can be performed on per-function basis:

- Every function is immediately dominated by root
- So, dominator tree oblivious to inter-procedural edges
- So, units & normal-form determined by local control-flow

Short digression: Trampolines

We realize tail-calls with *trampolines* for portable C code.

Basic trampoline

```
void trampoline(c){
  while(c ≠ NULL) {
    (f, x_1, ..., x_n) ← unpack c
    c ← f(x_1, ..., x_n)
  }
}
```

Using trampolines

Call	$f(\overline{x})$;	$\rightsquigarrow$	`trampoline(`$f(\overline{x})$`);`
Tail-call	`tail` $f(\overline{x})$;	$\rightsquigarrow$	`return (closure(`f, $\overline{x}$`));`
Return	`return;`	$\rightsquigarrow$	`return NULL;`

Runtime Interface

```
/* Closures */
typedef struct {...} closure_t;
closure_t* closure_make(closure_t* (*f)(τ x), τ x);
void closure_run(closure_t* c);

/* Modifiables */
typedef struct {...} modref_t;
void modref_init(modref_t *m);
void modref_write(modref_t *m, void *v);
closure_t* modref_read(modref_t *m, closure_t *c);

/* Allocation */
void* allocate(size_t n, closure_t *c);
```

Functions:

$$[\![f\ (\overline{\tau_x\ x})\ \{\overline{\tau_y\ y}\ ;\ \overline{b}\}]\!] = \texttt{closure\_t*}\ f(\overline{\tau_x\ x})\{\overline{\tau_y\ y;}\ [\![\overline{b}]\!]\}$$

Jumps:

$$[\![\texttt{goto}\ l]\!] = \texttt{goto}\ l;$$

$$[\![\texttt{tail}\ f(\overline{x})]\!] = \texttt{return (closure\_make(}f\texttt{,}\overline{x}\texttt{));}$$

Basic Blocks:

$$[\![\{l : \texttt{done}\}]\!] = \{l\texttt{:}\ \ \texttt{return NULL;}\}$$

$$[\![\{l : \texttt{cond}\ x\ j_1\ j_2\}]\!] = \{l\texttt{:}\ \ \texttt{if (}x\texttt{) \{}[\![j_1]\!]\texttt{\} else \{}[\![j_2]\!]\texttt{\}}\}$$

$$[\![\{l : c\ ;\ j\}]\!] = \{l\texttt{:}\ \ [\![c]\!];\ [\![j]\!]\}$$

$$\begin{array}{rcl} [\![\{l : x := \texttt{read}\ y\ ; & = & \{l\texttt{:}\ \ \texttt{closure\_t *}c\texttt{;} \\ \texttt{tail}\ f(x,\overline{z})\}]\!] & & \quad c\ \texttt{= closure\_make(}f\texttt{,NULL::}\overline{z}\texttt{);} \\ & & \quad \texttt{return (modref\_read(}y\texttt{,}c\texttt{));}\} \end{array}$$

Translation (1/2)

Functions:

$$[\![f\ (\overline{\tau_x\ x})\ \{\overline{\tau_y\ y}\ ;\ \overline{b}\}]\!] = \texttt{closure\_t*}\ f\texttt{(}\overline{\tau_x\ x}\texttt{)\{}\overline{\tau_y\ y\texttt{;}}\ [\![\overline{b}]\!]\texttt{\}}$$

Jumps:

$$[\![\texttt{goto}\ l]\!] = \texttt{goto}\ l\texttt{;}$$

$$[\![\texttt{tail}\ f(\overline{x})]\!] = \texttt{return}\ f\texttt{(}\overline{x}\texttt{);}$$

Basic Blocks:

$$[\![\{l : \mathrm{done}\}]\!] = \{l\texttt{:}\ \ \texttt{return NULL;}\}$$

$$[\![\{l : \mathrm{cond}\ x\ j_1\ j_2\}]\!] = \{l\texttt{:}\ \ \texttt{if (}x\texttt{) \{}[\![j_1]\!]\texttt{\} else \{}[\![j_2]\!]\texttt{\}}\}$$

$$[\![\{l : c\ ;\ j\}]\!] = \{l\texttt{:}\ \ [\![c]\!]\texttt{;}\ [\![j]\!]\}$$

$$\begin{aligned}[\![\{l : x := \texttt{read}\ y\ ; &= \{l\texttt{:}\ \ \texttt{closure\_t *}c\texttt{;}\\ \texttt{tail}\ f(x,\overline{z})\}]\!] &\qquad c\ \texttt{= closure\_make(}f\texttt{,NULL::}\overline{z}\texttt{);}\\ &\qquad \texttt{return (modref\_read(}y\texttt{,}c\texttt{));}\}\end{aligned}$$

Commands:

$$
\begin{array}{rcl}
[\![\texttt{nop}]\!] & = & \texttt{;} \\
[\![x := e]\!] & = & x \texttt{ = } [\![e]\!]; \\
[\![x[y] := e]\!] & = & x\texttt{[}y\texttt{] = } [\![e]\!]\texttt{;} \\
[\![\texttt{call } f(\overline{x})]\!] & = & \texttt{closure\_run(}f\texttt{(}\overline{x}\texttt{));} \\
[\![x := \texttt{alloc } y\ f\ \overline{z}]\!] & = & \texttt{closure\_t *}c\texttt{;} \\
 & & c \texttt{ = closure\_make(}f\texttt{,NULL::}\overline{z}\texttt{);} \\
 & & x \texttt{ = allocate(}y\texttt{,}c\texttt{);} \\
[\![x := \texttt{modref}()]\!] & = & [\![x := \texttt{alloc (sizeof(modref\_t))} \\
 & & \qquad\qquad \texttt{modref\_init } \langle\rangle]\!] \\
[\![\texttt{write } x := y]\!] & = & \texttt{modref\_write(}x\texttt{, }y\texttt{);}
\end{array}
$$

Benchmarks

List Primitives

filter, **map**, **reverse**, **minimum**, and **sum**

Sorting

quicksort and **mergesort**

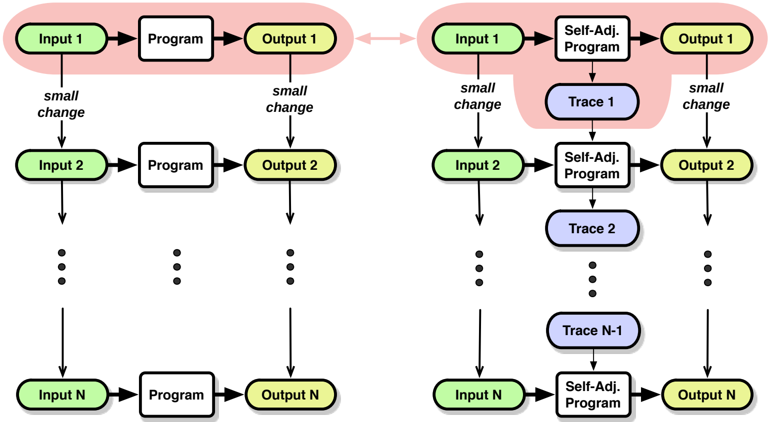
Computational Geometry

- **quickhull** finds convex hull
- **diameter** finds diameter of a set of points
- **distance** finds distance between two sets of points

Tree Algorithms

- **exprtree** evaluates an expression tree
- **tcon** performs **tree contraction**
  – general technique to compute properties of trees

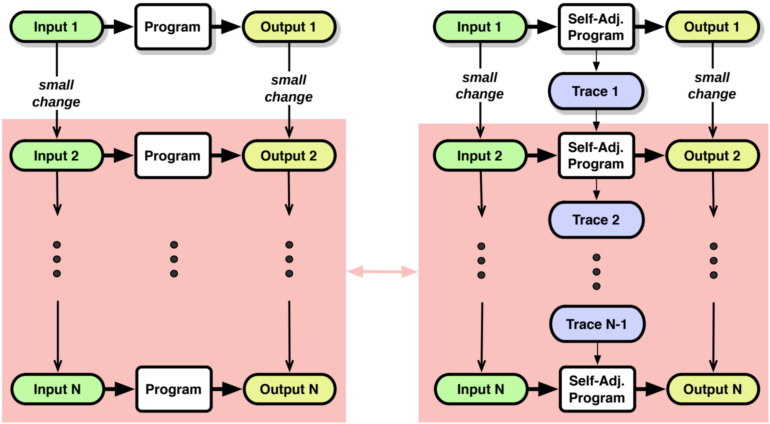
Overhead



Overhead

How much **slower** is the self-adjusting program when running "from-scratch"?

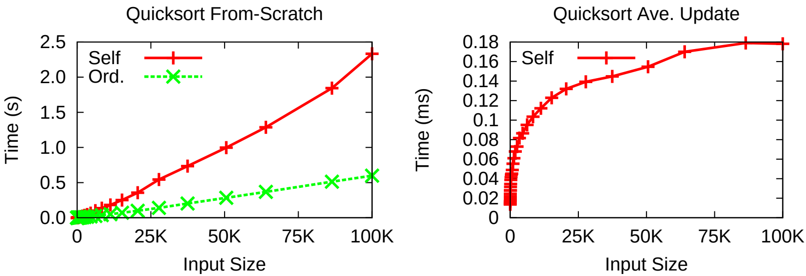
Speedup



Speedup

How much **faster** can the self-adjusting program update the output for a small change?

Evaluation Example: Quicksort



Overhead: about 4x
Speedup: 1.8×10^4

CEAL: Overhead & Speedup

		From-Scratch			Propagation		
Application	n	Cnv.	Self.	O.H.	Ave. Update	Speedup	Max Live
`filter`	1.0M	0.2	0.9	5.8	1.7×10^{-6}	9.3×10^{4}	292.4M
`map`	1.0M	0.4	1.0	2.5	1.9×10^{-6}	2.0×10^{5}	322.9M
`reverse`	1.0M	0.4	1.0	2.6	2.0×10^{-6}	2.0×10^{5}	322.9M
`minimum`	1.0M	0.2	1.3	7.7	4.0×10^{-6}	4.1×10^{4}	379.8M
`sum`	1.0M	0.2	1.3	7.9	6.4×10^{-5}	2.5×10^{3}	379.7M
`quicksort`	100.0K	0.6	2.3	3.9	1.8×10^{-4}	3.4×10^{3}	834.3M
`quickhull`	100.0K	0.2	1.2	7.0	1.3×10^{-4}	1.3×10^{3}	649.4M
`diameter`	100.0K	0.2	1.1	6.4	1.2×10^{-4}	1.4×10^{3}	642.8M
`exptrees`	1.0M	3.8	4.9	1.3	2.5×10^{-4}	1.5×10^{4}	517.9M
`mergesort`	100.0K	0.9	3.6	3.8	9.6×10^{-5}	9.9×10^{3}	1.4G
`distance`	100.0K	0.2	1.1	6.5	1.8×10^{-4}	9.5×10^{2}	501.6M
`tcon`	100.0K	1.2	4.1	3.5	3.2×10^{-4}	3.7×10^{3}	899.6M

- **Average Overhead**: 5x
- **Average Speedup**: 9.2×10^4 ($n = 1\text{M}$), 3.4×10^3 ($n = 100\text{k}$),

CEAL *vs* SaSML: Summary

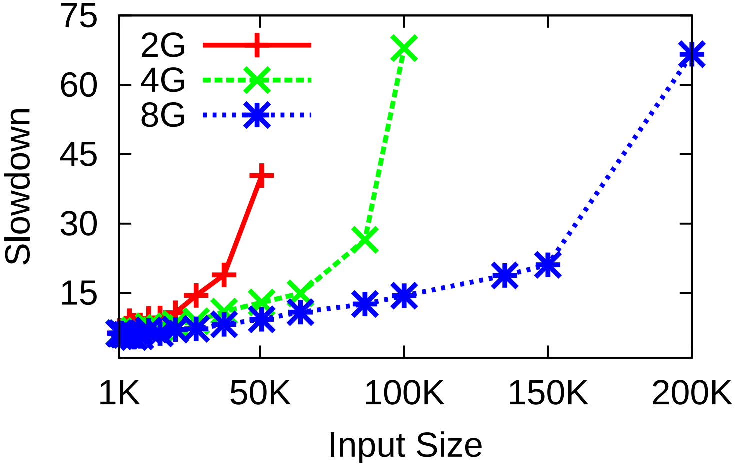
Measurements with SaSML

	From-Scratch			Propagation		
App.	Cnv.	Self.	S.D.	Ave. Update	S.D.	Max Live
`filter`	0.1	6.9	7.7	8.7×10^{-6}	5.1	1398M
`map`	0.1	7.8	7.8	1.1×10^{-5}	5.8	1593M
`reverse`	0.1	6.7	6.7	9.3×10^{-6}	4.7	1516M
`minimum`	0.1	5.1	3.9	3.0×10^{-5}	7.5	1168M
`sum`	0.1	5.1	3.9	1.7×10^{-4}	2.7	1187M
`quicksort`	0.2	52	22.6	1.7×10^{-3}	9.4	3950M
`quickhull`	0.7	5.1	4.2	3.3×10^{-4}	2.5	774M
`diameter`	0.9	5.2	4.7	3.7×10^{-4}	3.1	943M

SaSML vs CEAL in summary

- **From-Scratch** slowdown: 4-23x, (8x on average)
- **Change propagation** slowdown: 3-9x (5x on average)
- **Max live**: up to 5x larger (3.5x on average)

CEAL *vs* SaSML: Quicksort Slowdown



Change prop. "slowdown": SaSML time divided by CEAL time.

Concluding remarks

In Summary

- CEAL is a C-based language for self-adjusting computation
- CEAL can be compiled directly to (portable) C code:
  - **Normalization** transform for tracing, re-evaluation and reuse.
  - **Translation** to C uses trampolines to implement tail jumps.
- Performance results are promising.

Future work

- Separate `memo` primitive
- Eliminate need for explicit allocation init funs
- Automatically minimize "read scopes"

Thanks, Questions

Thank You!

Questions?