Type-Directed Automatic Incrementalization

Yan Chen
Joshua Dunfield Umut A. Acar

Max Planck Institute for Software Systems

June 12, 2012

Incremental Problems

An example: Computing maximum

- **Input:** 3, 5, 8, 2, 10, 4, 9, 1
- **Output:** Max $= 10$

Efficiency

- **Linear scan:** $O(n)$

Incremental Problems

An example: Computing maximum

- **Input:** 3, 5, 8, 2, ~~10,~~ 4, 9, 1
- **Output:** Max $=$ ~~10~~ 9

Efficiency

- **Linear scan:** $O(n)$
- **Priority queue:** $O(\log n)$

Motivation: Incremental algorithms are complex

Examples

Problem	Static	Incremental
Maximum	1950s: $O(n)$	1964: $O(\log n)$
Mesh Refinement	1989: $O(n)$	2011: $O(\log n)$
2D Convex Hull	1972: $O(n \log n)$	2002: $O(\log n)$
$\vdots$		
Compilation	Whole-program	Separate

Challenge

Static versus Incremental

Static Algorithms $\xrightarrow[\text{10+ years of research}]{}$ Incremental Algorithms

Challenge: Automatic Incrementalization

Can we incrementalize a static algorithm?

Example

Incrementalization with Dependency Graphs

Code

```
fun sumSq (x, y) =
  let
    val x2 = x * x
    val y2 = y * y
  in
    x2 + y2
  end
```

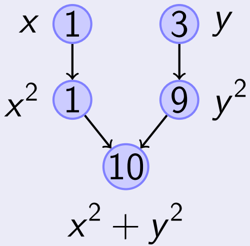
Example

Incrementalization with Dependency Graphs

Code

```
fun sumSq (x, y) =
  let
    val x2 = x * x
    val y2 = y * y
  in
    x2 + y2
  end
```

Dep. Graph



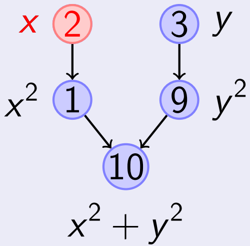
Example

Incrementalization with Dependency Graphs

Code

```
fun sumSq (x, y) =
  let
    val x2 = x * x
    val y2 = y * y
  in
    x2 + y2
  end
```

Dep. Graph



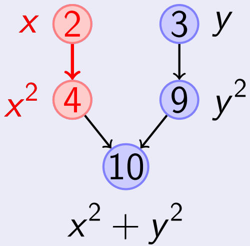
Example

Incrementalization with Dependency Graphs

Code

```
fun sumSq (x, y) =
  let
    val x2 = x * x
    val y2 = y * y
  in
    x2 + y2
  end
```

Dep. Graph



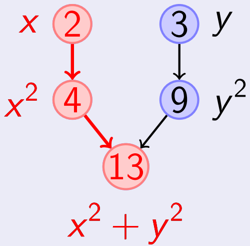
Example

Incrementalization with Dependency Graphs

Code

```
fun sumSq (x, y) =
  let
    val x2 = x * x
    val y2 = y * y
  in
    x2 + y2
  end
```

Dep. Graph



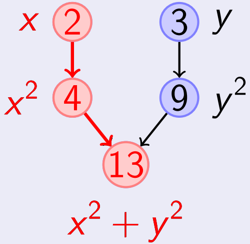
Explicit Self-Adjusting Computation

Rewrite program to construct dependency graph

Code

```
fun sumSq(x:int     ,y:int) =
let
  val x2 =
                      x * x
  val y2 = y * y
in

              x2  + y2
end
```

Dep. Graph



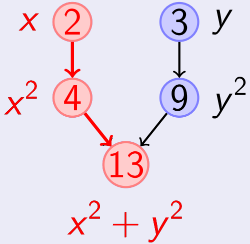
Explicit Self-Adjusting Computation

Rewrite program to construct dependency graph

Code

```
fun sumSq(x:int mod,y:int) =
let
  val x2 = mod (read x as x'
              in write (x'* x'))
  val y2 = y * y
in
  mod (read x2 as x2' in
       write (x2' + y2))
end
```

Dep. Graph



Limitations of Explicit Self-Adjusting Computation

- The explicit library is not a natural way of programming.

```
fun sumSq (x:int mod, y:int) =
  let
    val x2 = mod (read x as x' in
                  write (x' * x'))
    val y2 = y * y
  in
    mod (read x2 as x2' in
         write (x2' + y2))
   end
```

Limitations of Explicit Self-Adjusting Computation

- The explicit library is not a natural way of programming.
- Efficiency is highly sensitive to program details.

```
fun sumSq (x:int mod, y:int) =
  let
    val x2 = mod (read x as x' in
                   write (x' * x'))
    val y2 = y * y
  in
    mod (read x2 as x2' in
         write (x2' + y2))
   end
```

Limitations of Explicit Self-Adjusting Computation

- The explicit library is not a natural way of programming.
- Efficiency is highly sensitive to program details.

```
fun sumSq (x:int mod, y:int) =
  let
    val x2 = mod (read x as x' in
                 write (x' * x' + y * y))

  in


    x2
  end
```

Limitations of Explicit Self-Adjusting Computation

- The explicit library is not a natural way of programming.
- Efficiency is highly sensitive to program details.
- Different requirements lead to different functions.

```
fun sumSq (x:int, y:int mod) =
  let
    val x2 = x * x
    val res = mod (read y as y' in
                     write (x2 + y' * y'))
  in

    res
  end
```

Limitations of Explicit Self-Adjusting Computation

- The explicit library is not a natural way of programming.
- Efficiency is highly sensitive to program details.
- Different requirements lead to different functions.
- Function rewriting can spread to large amounts of code.

```
fun sumSq (x:int, y:int mod) =
  let
    val x2 = x * x
    val res = mod (read y as y' in
                    write (x2 + y' * y'))
  in

    res
  end
```

This Talk: Bridge the Gap

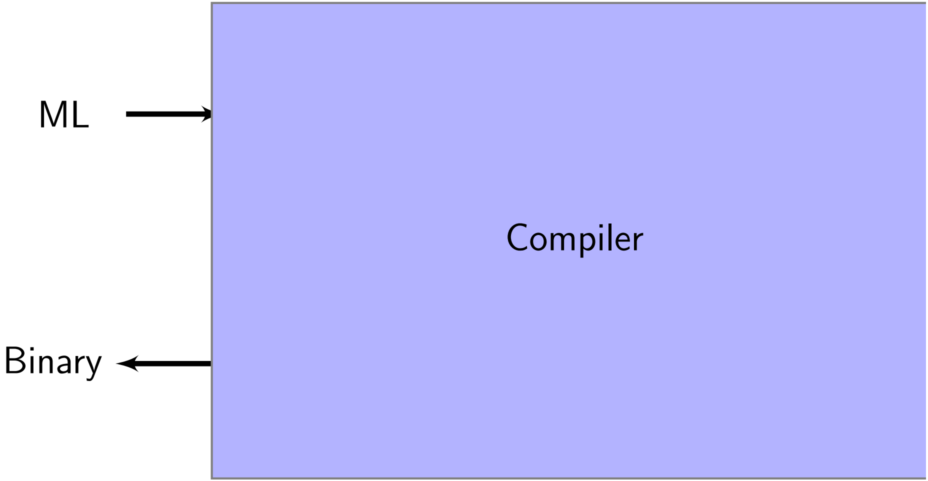
ML Code

```
fun sumSq ( x ,  y ) =
let
  val x2 = x * x
  val y2 = y * y
 in
  x2 + y2
 end
```

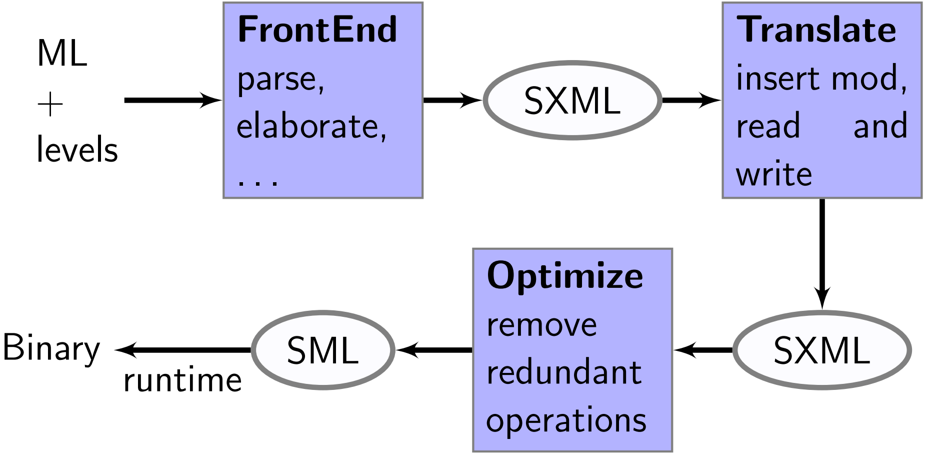
Explicit Self-Adjusting Code

```
fun sumSq (x:int mod, y:int) =
let
  val x2 = mod (read x as x' in
                    write (x' * x'))
  val y2 = y * y
in
  mod (read x2 as x2' in
         write (x2' + y2))
end
```

Compiler Architecture



Compiler Architecture



SML extended with level types

Specify SML program with levels $\mathcal{C}/\mathcal{S}$

Examples

```
type intC = int C

datatype list S = nil
                | cons of int C * list S

datatype list C = nil
                | cons of int S * list C
```

FrontEnd

- Propagate level types for all subterms
- Supports full Standard ML and its module system

Code

```
fun sumSq (x :int^C, y :int^S) : int =
let
  val x2 :int = x * x
  val y2 :int = y * y
  val res :int = x2 + y2
in res end
```

FrontEnd

- Propagate level types for all subterms
- Supports full Standard ML and its module system

Code

```
fun sumSq (x :int^C, y :int^S) : int =
let
  val x2 :int^C = x * x
  val y2 :int = y * y
  val res :int = x2 + y2
in res end
```

FrontEnd

- Propagate level types for all subterms
- Supports full Standard ML and its module system

Code

$\texttt{fun sumSq (x:int}^{\mathcal{C}}\texttt{, y:int}^{\mathcal{S}}\texttt{) : int =}$

$\texttt{let}$

$\quad\texttt{val x2:int}^{\mathcal{C}}\texttt{ = x * x}$

$\quad\texttt{val y2:int}^{\mathcal{S}}\texttt{ = y * y}$

$\quad\texttt{val res:int = x2 + y2}$

$\texttt{in res end}$

FrontEnd

- Propagate level types for all subterms
- Supports full Standard ML and its module system

Code

fun sumSq (x : $\text{int}^{\mathcal{C}}$, y : $\text{int}^{\mathcal{S}}$) : int =
let
  val x2 : $\text{int}^{\mathcal{C}}$ = x * x
  val y2 : $\text{int}^{\mathcal{S}}$ = y * y
  val res : $\text{int}^{\mathcal{C}}$ = x2 + y2
in res end

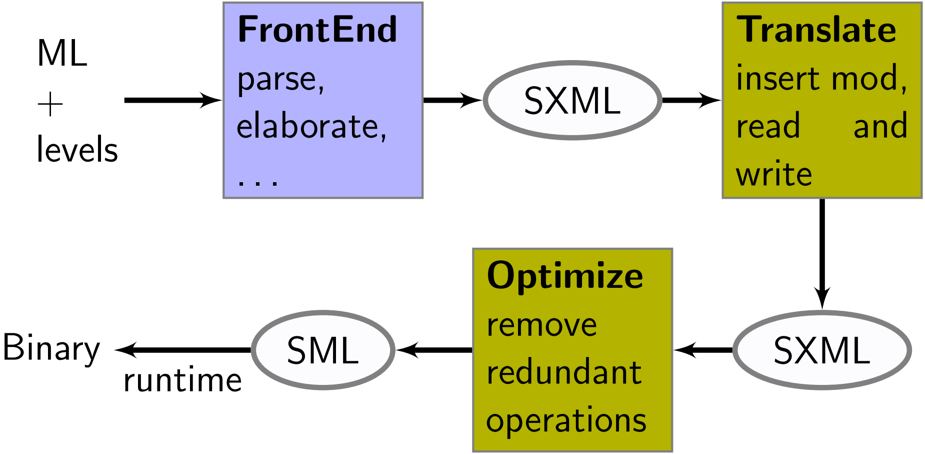
FrontEnd

- Propagate level types for all subterms
- Supports full Standard ML and its module system

Code

fun sumSq (x :$\texttt{int}^{\mathcal{C}}$, y :$\texttt{int}^{\mathcal{S}}$) : $\texttt{int}^{\mathcal{C}}$ =
let
  val x2 :$\texttt{int}^{\mathcal{C}}$ = x * x
  val y2 :$\texttt{int}^{\mathcal{S}}$ = y * y
  val res :$\texttt{int}^{\mathcal{C}}$ = x2 + y2
in res end

Compiler Architecture



Translate

Translate the code locally using level types

Translation Example

Typing Environment Γ: x2:int$^{\mathcal{C}}$, y2:int$^{\mathcal{S}}$

$$\Gamma \vdash \ \mathsf{val}\ \boxed{\mathsf{res}} = \boxed{\mathsf{x2}} + \boxed{\mathsf{y2}}\ \mathsf{in}\ \boxed{\mathsf{res}} : \mathsf{int}^{\mathcal{C}}$$

$$\underset{\mathcal{S}}{\hookrightarrow}$$

Translate

Translate the code locally using level types

Translation Example

Typing Environment Γ: x2:int$^{\mathcal{C}}$, y2:int$^{\mathcal{S}}$

$$\Gamma \vdash \quad \text{val } \boxed{\text{res}} = \boxed{\text{x2}} + \boxed{\text{y2}} \text{ in } \boxed{\text{res}} : \text{int}^{\mathcal{C}}$$

$\underset{\mathcal{S}}{\hookrightarrow}$

```
val res =
                        y2
```

Translate

Translate the code locally using level types

Translation Example

Typing Environment Γ: x2:int$^{\mathcal{C}}$, y2:int$^{\mathcal{S}}$

$$\Gamma \vdash \quad \text{val } \boxed{\text{res}} = \boxed{\text{x2}} + \boxed{\text{y2}} \text{ in } \boxed{\text{res}} : \text{int}^{\mathcal{C}}$$

$\underset{\mathcal{S}}{\hookrightarrow}$

```
val res =     read x2 as x2' in
                     x2' + y2
```

Translate

Translate the code locally using level types

Translation Example

Typing Environment Γ: x2:int$^{\mathcal{C}}$, y2:int$^{\mathcal{S}}$

$$\Gamma \vdash \text{val } \boxed{\text{res}} = \boxed{\text{x2}} + \boxed{\text{y2}} \text{ in } \boxed{\text{res}} : \text{int}^{\mathcal{C}}$$

$\underset{\mathcal{S}}{\hookrightarrow}$

```
val res =        read x2 as x2' in
                 write (x2' + y2)
```

Translate

Translate the code locally using level types

Translation Example

Typing Environment Γ: x2:int$^{\mathcal{C}}$, y2:int$^{\mathcal{S}}$, res:int$^{\mathcal{C}}$

$$\Gamma \vdash \; \text{val } \boxed{\text{res}} = \boxed{\text{x2}} + \boxed{\text{y2}} \text{ in } \boxed{\text{res}} : \text{int}^{\mathcal{C}}$$

$\underset{\mathcal{S}}{\hookrightarrow}$

```
val res = mod (read x2 as x2' in
                 write (x2' + y2))
```

Translate

Translate the code locally using level types

Translation Example

Typing Environment Γ: x2:$\text{int}^{\mathcal{C}}$, y2:$\text{int}^{\mathcal{S}}$, res:$\text{int}^{\mathcal{C}}$

$$\Gamma \vdash \;\; \text{val } \boxed{\text{res}} = \boxed{\text{x2}} + \boxed{\text{y2}} \text{ in } \boxed{\text{res}} : \text{int}^{\mathcal{C}}$$

$$\underset{\mathcal{S}}{\hookrightarrow}$$

```
val res = mod (read x2 as x2' in
                    write (x2' + y2))
in
```

Translate

Translate the code locally using level types

Translation Example

Typing Environment Γ: x2:int$^{\mathcal{C}}$, y2:int$^{\mathcal{S}}$, res:int$^{\mathcal{C}}$

$$\Gamma \vdash \text{val } \boxed{\text{res}} = \boxed{\text{x2}} + \boxed{\text{y2}} \text{ in } \boxed{\text{res}} : \text{int}^{\mathcal{C}}$$

$$\underset{\mathcal{S}}{\hookrightarrow}$$

```
val res = mod (read x2 as x2' in
                 write (x2' + y2))
in  mod (read res as r in
         write r)
```

Optimize

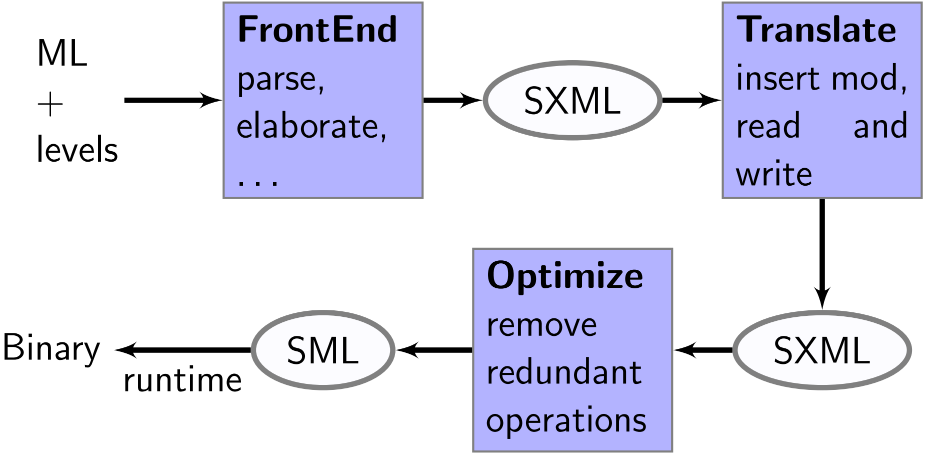
Rewriting Rules

$$\begin{array}{rcll}
\textbf{read}\ (\textbf{mod}\ e)\ \textbf{as}\ x'\ \textbf{in}\ \textbf{write}(x') & \longrightarrow & e & (1)\\
\textbf{mod}\ (\textbf{read}\ e\ \textbf{as}\ x'\ \textbf{in}\ \textbf{write}(x')) & \longrightarrow & e & (2)\\
\textbf{read}(\textbf{mod}\ (\textbf{let}\ r = e_1\ \textbf{in}\ \textbf{write}(r))) & & & \\
\textbf{as}\ x'\ \textbf{in}\ e_2 \longrightarrow \textbf{let}\ x' = e_1 & \textbf{in} & e_2 & (3)
\end{array}$$

Correctness and Efficiency

- Rewriting rules are terminating and confluent
- Reduce the execution time for self-adjusting programs by up to 60%

Compiler Architecture

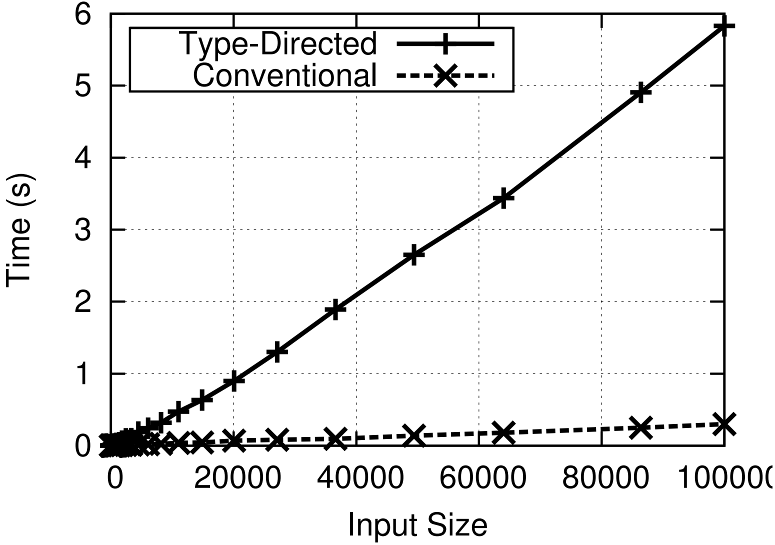


Summary of Benchmark Timings

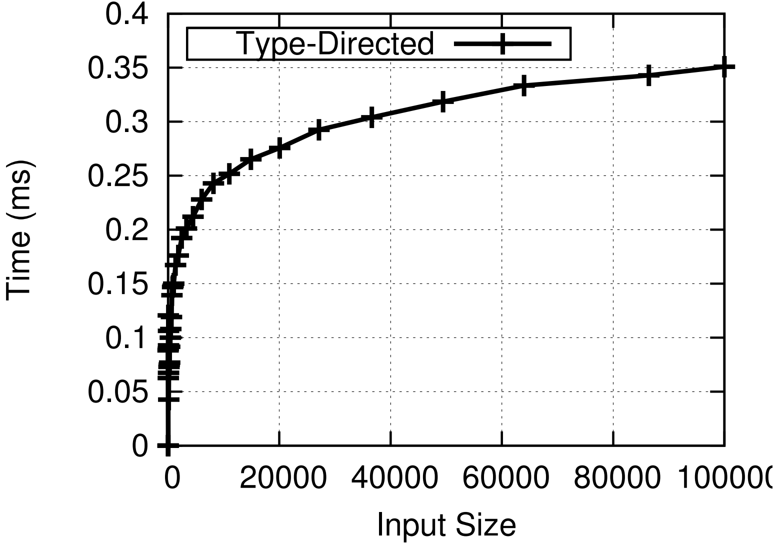
Application	*Conv.*	*S.A.*	*Propagation*	*O.H.*	*Speedup*
map(10^6)	0.05	0.83	1.1×10^{-6}	16.7	4.6×10^{4}
filter(10^6)	0.04	1.25	1.4×10^{-6}	27.7	3.2×10^{4}
split(10^6)	0.14	1.63	3.2×10^{-6}	11.6	4.4×10^{4}
msort(10^5)	0.30	5.83	3.5×10^{-4}	19.5	850.92
qsort(10^5)	0.05	3.40	4.9×10^{-4}	64.2	108.17
vec-reduce(10^6)	0.05	0.26	4.4×10^{-6}	5.5	1.1×10^{4}
vec-mult(10^6)	0.18	1.10	6.7×10^{-6}	5.9	2.8×10^{4}
mat-vec-mult(10^3)	0.17	0.81	1.4×10^{-5}	4.6	1.3×10^{4}
mat-add(10^3)	0.10	0.36	4.9×10^{-7}	3.7	2.0×10^{5}
transpose(10^4)	2.14	2.15	5.1×10^{-8}	1.0	4.2×10^{7}

- Overhead (O.H.) $= \dfrac{\text{Self-Adjusting Run (S.A.)}}{\text{Conventional Run}}$
- Speedup $= \dfrac{\text{Conventional Run}}{\text{Change propagation}}$

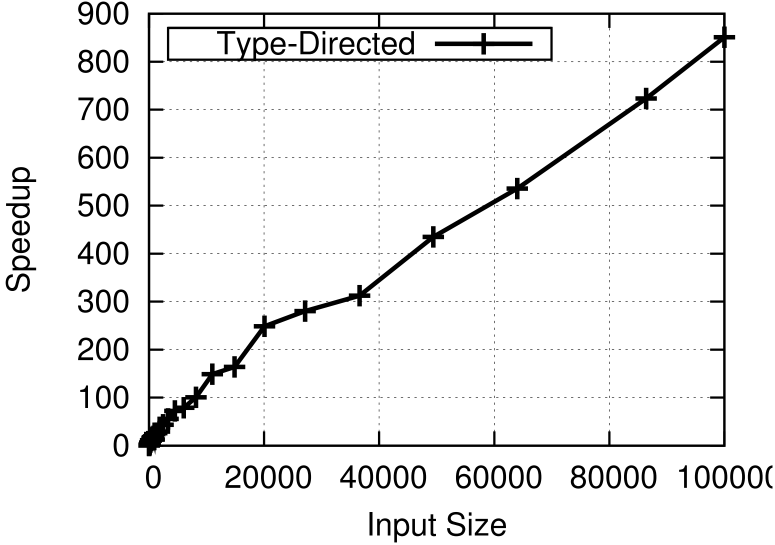
Merge Sort — Complete Run



Merge Sort — Change Propagation



Merge Sort — Speedup



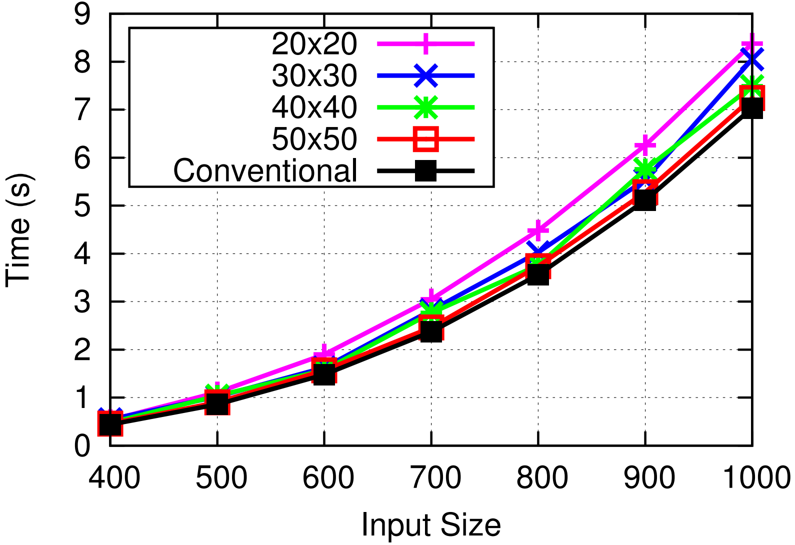
Matrix Multiplication — Granularity Control

```
type matrix = ((int C) vector) vector

type block = ((int S) vector) vector
type matrix = ((block C) vector) vector
```

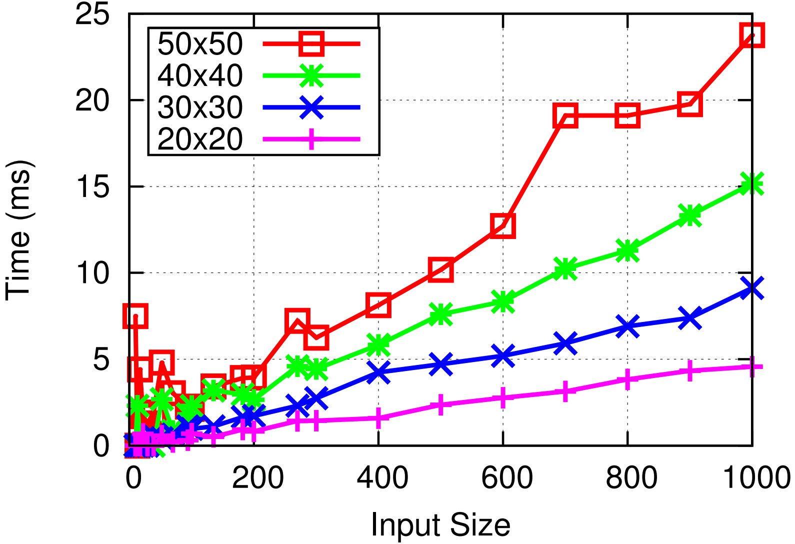
Matrix Multiply: Overhead and Speedup

Application	Overhead	Speedup
matrix (400)	8.5	1.8×10^3
block20 (10^3)	1.2	1.5×10^3

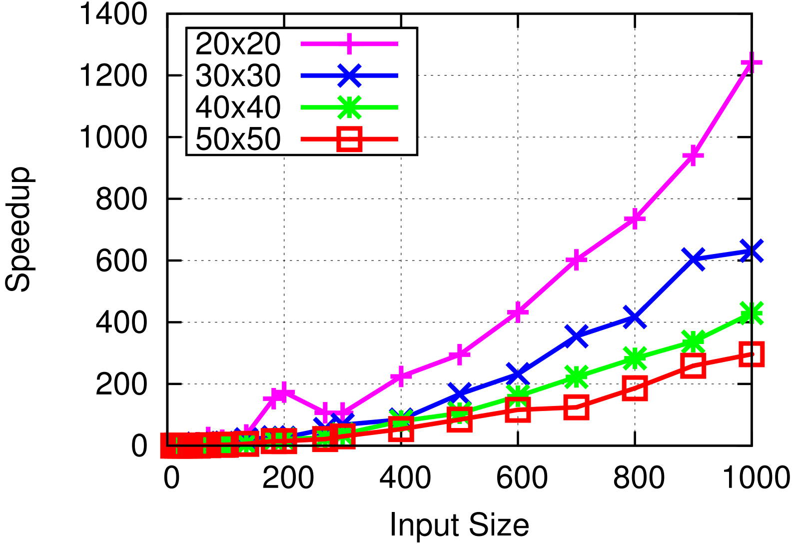
Matrix Multiplication — Complete Run



Matrix Multiplication — Change Propagation



Matrix Multiplication — Speedup



Ray Tracer

- Take an off-the-shelf, non-incremental ray tracer
- Change the surface property of a scene: color, reflective, diffuse

$$\texttt{datatype Surface = T of } \{\texttt{ambient: Vector.t } \mathcal{C}\texttt{,}$$
$$\texttt{diffuse: Vector.t } \mathcal{C}\texttt{,}$$
$$\texttt{transmit :real } \mathcal{C}\texttt{,...}\}$$
$$\texttt{type background = Vector.t } \mathcal{C}$$

Not stable: small input change $\longrightarrow$ large output change

Ray Tracer — Source code

Code Size & Annotations

- 3896 lines of code. 7 lines annotated.

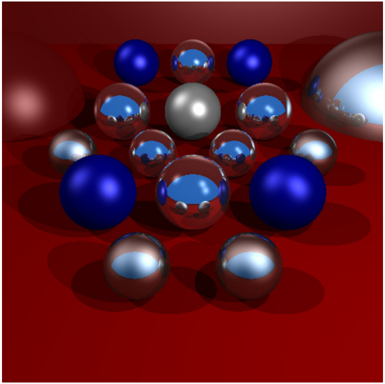
Code “diff”

```
  fun refractRay (dir, normDir, refrIndex) =
     let
!       val refrIndex = refrIndex
        val dotp = ~(Vector.dot (dir, normDir))
!       val (norm, k, nr) =
           if (dotp < 0.0)
!             then (Vector.scale (normDir, ~1.0), ~dotp, 1.0/refrIndex)
!          else (normDir, dotp, refrIndex) (* trans. only with air *)
!       val disc = 1.0 - nr*nr*(1.0-k*k)
!
     in

        if (disc < 0.0)
!          then NONE
!       else let
!
               val t = nr * k - (Math.sqrt disc)
            in
!              SOME (Vector.add (Vector.scale (norm, t),
!                                Vector.scale (dir, nr)))
!           end
     end

  (*
  % Transmit a ray through an object
  *)
  fun transmitColor (pos, dir, normDir, refrDir, intens, contrib) =
     let
!       val contrib' = Vector.mul (intens, contrib)
     in

        if Vector.isZero contrib'
!          then (0.0,0.0,0.0) (* cutoff *)
!       else (case refrDir of
!                NONE => (0.0,0.0,0.0)
!
              | SOME refrDir =>
                   let
                      (* need to offset just a bit *)
                      val ray' = {pos = Vector.add (pos, Vector.scale (refrDir, epsilon)),
                                  dir = refrDir}
                      val color' = traceAndShade (ray', contrib')
!                  in
!                     Vector.mul (color', intens)
!                  end)
     end

  (*
  % Reflect a ray from an object
  *)
  and reflectColor (pos, reflDir, intens, contrib) =
```

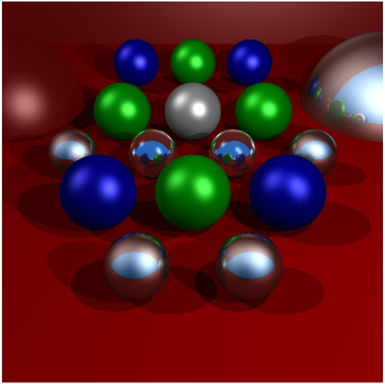
```
  fun refractRay (dir, normDir, refrIndex) =
     let
!       val eq = fn ((a1,a2,a3),(b1,b2,b3)) => (MLton.eq (a1,b1)) andalso (MLton.eq (a2,b2))
        val dotp = ~(Vector.dot (dir, normDir))
!       val result = C.modref (refrIndex ==> (fn refrIndex =>
           if (dotp < 0.0)
!             then let val res = (Vector.scale (normDir, ~1.0), ~dotp, 1.0/refrIndex)
!                  in C.write' eq res end
!          else C.write' eq (normDir, dotp, refrIndex))) (* trans. only with air *)
!       val disc = C.modref (result ==> (fn (_, k, nr) => C.write' MLton.eq (1.0 - nr*nr*(1.0
     in
+       C.modref (disc ==> (fn disc =>
        if (disc < 0.0)
!          then C.write' MLton.eq NONE
!       else result ==> (fn (norm, k, nr) =>
!           let
               val t = nr * k - (Math.sqrt disc)
            in
!              C.write' MLton.eq (SOME (Vector.add (Vector.scale (norm, t),
!                                                  Vector.scale (dir, nr))))
!           end)))
     end

  (*
  % Transmit a ray through an object
  *)
  fun transmitColor (pos, dir, normDir, refrDir, intens, contrib) =
     let
!       val contrib' = C.modref (intens ==> (fn intens => write (Vector.mul (intens, contrib)
     in
+       C.modref (contrib' ==> (fn contrib' =>
        if Vector.isZero contrib'
!          then write (0.0,0.0,0.0) (* cutoff *)
!       else (refrDir ==> (fn refrDir =>
!             case refrDir of
!                NONE => write (0.0,0.0,0.0)
              | SOME refrDir =>
                   let
                      (* need to offset just a bit *)
                      val ray' = {pos = Vector.add (pos, Vector.scale (refrDir, epsilon)),
                                  dir = refrDir}
                      val color' = traceAndShade (ray', contrib')
!                  in color' ==> (fn color' => intens ==> (fn intens =>
!                        write (Vector.mul (color', intens))))
!                  end))))
     end

  (*
  % Reflect a ray from an object
  *)
  and reflectColor (pos, reflDir, intens, contrib) =
```

Ray Tracer

Reflective surface



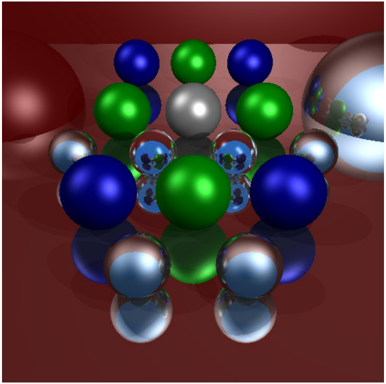
Green diffuse surface



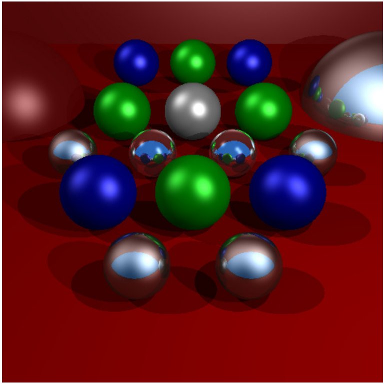
Change	Output Diff	Speedup
reflective $\longrightarrow$ diffuse	8.43%	4.29
diffuse $\longrightarrow$ reflective	8.43%	2.44

Ray Tracer

Reflective background



Diffuse background (red)



Change	Output Diff	Speedup
reflective $\rightarrow$ diffuse	57.22%	1.34
diffuse $\rightarrow$ reflective	57.22%	0.22

Summary

Type-Direct Automatic Incrementalization

- High-level language for incremental computation
- Incrementalization with simple annotations
- Translation generates self-adjusting code
- Optimizations improve efficiency
- Yields asymptotically and practically efficient binaries
- Empirical analysis of space usage

Future Works

- Fully automatic type inference and granularity control
- Memoization
- Experiments on large software projects