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

<table>
<thead>
<tr>
<th>Problem</th>
<th>Static</th>
<th>Incremental</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum</td>
<td>1950s: $O(n)$</td>
<td>1964: $O(\log n)$</td>
</tr>
<tr>
<td>Mesh Refinement</td>
<td>1989: $O(n)$</td>
<td>2011: $O(\log n)$</td>
</tr>
<tr>
<td>2D Convex Hull</td>
<td>1972: $O(n \log n)$</td>
<td>2002: $O(\log n)$</td>
</tr>
<tr>
<td>Compilation</td>
<td>Whole-program</td>
<td>Separate</td>
</tr>
</tbody>
</table>
Challenge

Static versus Incremental

Static Algorithms 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

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

Dep. Graph

```
x 1       3  y
x^2 1       9  y^2
    10
```

```
x^2 + y^2
```
Example

Incrementalization with Dependency Graphs

Code

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

Dep. Graph

```
x
v 10

x^2 + y^2
```

Yan Chen

Type-Directed Automatic Incrementalization
Incrementalization with Dependency Graphs

**Examples**

Code

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

Dep. Graph

```
x 2

x^2 4

y 3

x^2 + y^2
```

Yan Chen

Type-Directed Automatic Incrementalization
Example

Incrementalization with Dependency Graphs

Code

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

Dep. Graph

```
x [2]  
\downarrow
x^2 [4]  
\downarrow
13

\downarrow
y [3]  
\downarrow
y^2 [9]  

x^2 + y^2
```
Explicit Self-Adjusting Computation

Rewrite program to construct dependency graph

Code

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

Dep. Graph

```
\[
\begin{align*}
  x^2 & \rightarrow 4 \rightarrow 13 \\
  y^2 & \rightarrow 3
\end{align*}
\]

\[
x^2 + y^2 
\]
Explicit Self-Adjusting Computation

Rewrite program to construct dependency graph

**Code**

```haskell
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**

```
x 2 → 4 → 13
x^2

3 → 9 → 13
y^2

x^2 + y^2
```
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.

```ocaml
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
```
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.

```ml
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.

```plaintext
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**

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

**Explicit Self-Adjusting Code**

```ml
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

ML

Compiler

Binary

Yan Chen

Type-Directed Automatic Incrementalization
Compiler Architecture

ML + levels

FrontEnd
parse, elaborate, ...

SXML

Translate
insert mod, read and write

Optimize
remove redundant operations

Binary runtime

SML

Yan Chen Type-Directed Automatic Incrementalization
SML extended with level types

Specify SML program with levels $C/S$

Examples

type int$C = int \ C$

datatype list $S = nil$
| cons of int $C \ast list \ S$

datatype list $C = nil$
| cons of int $S \ast list \ C$
FrontEnd

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

Code

```ml
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 (\(x:\text{int}^C\), \(y:\text{int}^S\)) : int =
let
  val \(x^2:\text{int}^C\) = \(x \ast x\)
  val \(y^2:\text{int}\) = \(y \ast y\)
  val res : int = \(x^2 + y^2\)
  in  res  end
FrontEnd

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

Code

```ml
fun sumSq (x:int\^c, y:int\^S) : int =
let
  val x2:int\^c = x * x
  val y2:int\^S = y * y
  val res:int = x2 + y2
in res end
```
Propagate level types for all subterms
Supports full Standard ML and its module system

fun sumSq (x:int\textsuperscript{C}, y:int\textsuperscript{S}) : int =
  let
    val x2 : int\textsuperscript{C} = x * x
    val y2 : int\textsuperscript{S} = y * y
    val res : int\textsuperscript{C} = x2 + y2
  in res end
Propagate level types for all subterms
Supports full Standard ML and its module system

fun sumSq (\(x: \text{int}^c\), \(y: \text{int}^s\) : \(\text{int}^c\) =
let
  val \(x2: \text{int}^c\) = \(x \times x\)
  val \(y2: \text{int}^s\) = \(y \times y\)
  val \(res: \text{int}^c\) = \(x2 + y2\)
in  res  end
Compiler Architecture

ML + levels

FrontEnd
parse, elaborate, ...

SXML
Translate
insert mod, read and write

SML
Optimize
remove redundant operations

Binary runtime

Yan Chen Type-Directed Automatic Incrementalization 12
Translate

Translate the code locally using level types

Translation Example

Typing Environment $\Gamma: x2: \text{int}^C, y2: \text{int}^S$

$$\Gamma \vdash \text{val} \; \text{res} = \text{int}^{\text{C}}$$
Translate

Translate the code locally using level types

**Translation Example**

Typing Environment $\Gamma$: $x_2: \text{int}^C$, $y_2: \text{int}^S$

\[
\Gamma \vdash \text{val} \ res = x_2 + y_2 \quad \text{in} \quad res : \text{int}^C
\]

\[
\text{val res } =
\]

\[
\rightarrow
\]

\[
y_2
\]
Translate

Translate the code locally using level types

Translation Example

Typing Environment $\Gamma$: $x_2: \text{int}^C$, $y_2: \text{int}^S$

$\Gamma \vdash \text{val res} = x_2 + y_2 \quad \text{in} \quad \text{res} : \text{int}^C$

$\text{val res} = \text{read } x_2 \text{ as } x_2' \quad \text{in} \quad x_2' + y_2$

Yan Chen  
Type-Directed Automatic Incrementalization  
13
Translate

Translate the code locally using level types

Translation Example

Typing Environment $\Gamma$: $x2: \text{int}^C$, $y2: \text{int}^S$

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

$\rightarrow$

-val res =

  read x2 as x2’ in

  write (x2’ + y2)
Translate the code locally using level types

**Translation Example**

Typing Environment $\Gamma$: $x_2: \text{int}^C$, $y_2: \text{int}^S$, $\text{res: int}^C$

\[
\Gamma \vdash \text{val } \text{res} = x_2 + y_2 \text{ in } \text{res : int}^C
\]

\[
\text{val res} = \text{mod (read } x_2 \text{ as } x_2' \text{ in write (} x_2' + y_2 \text{))}
\]
Translate

Translate the code locally using level types

Translation Example

Typing Environment $\Gamma$: $x_2$:$\text{int}^C$, $y_2$:$\text{int}^S$, $\text{res}$:$\text{int}^C$

\[
\Gamma \vdash \text{val } \text{res} = x_2 + y_2 \text{ in } \text{res} : \text{int}^C
\]

\[
\text{val res = mod } (\text{read } x_2 \text{ as } x_2' \text{ in } \text{write } (x_2' + y_2))
\]
Translate the code locally using level types

Translation Example

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

\[
\Gamma \vdash \text{val } \fbox{res} = \fbox{x2} + \fbox{y2} \text{ in } \fbox{res} : \text{int}^C
\]

\[
\text{val } \fbox{res} = \text{mod } (\text{read } x2 \text{ as } x2' \text{ in } \text{write } (x2' + y2))
\]

\[
\xrightarrow{s} \text{ in } \text{mod } (\text{read } \text{res as } r \text{ in } \text{write } r)
\]
Optimize

| Rewriting Rules | |
|-----------------|[
| **read (mod e) as x’ in write(x’) → e** (1) | |
| **mod (read e as x’ in write(x’)) → e** (2) | |
| **read (mod (let r = e₁ in write(r)) as x’ in e₂ → let x’ = e₁ in e₂** (3) | |

**Correctness and Efficiency**

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

ML + levels

FrontEnd
parse, elaborate, ...

SXML
Translate
insert mod, read and write

Optimize
remove redundant operations

SXML

Binary runtime

SML

Yan Chen 
Type-Directed Automatic Incrementalization
### Summary of Benchmark Timings

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

- Overhead (O.H.) = \( \frac{\text{Self-Adjusting Run (S.A.)}}{\text{Conventional Run}} \)
- Speedup = \( \frac{\text{Conventional Run}}{\text{Change propagation}} \)
Merge Sort — Complete Run

![Graph showing comparison between Type-Directed and Conventional methods for Merge Sort.

- Time (s) on the y-axis.
- Input Size on the x-axis, ranging from 0 to 100000.
- Two lines represent the methods:
  - Type-Directed (solid line)
  - Conventional (dashed line)

The Type-Directed method consistently outperforms the Conventional method across all input sizes, as indicated by the steeper slope of the Type-Directed line. This suggests that the Type-Directed approach is more efficient in terms of computational time.

Yan Chen
Type-Directed Automatic Incrementalization
17
Merge Sort — Change Propagation

![Graph showing time (ms) vs input size for Type-Directed change propagation]

- **Time (ms)**
- **Input Size**

Yan Chen | Type-Directed Automatic Incrementalization | 18
Merge Sort — Speedup

![Graph showing speedup vs. input size for Type-Directed algorithm.

- **Yan Chen**  
  Type-Directed Automatic Incrementalization

---

*Graph credits: [Source link]*
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

<table>
<thead>
<tr>
<th>Application</th>
<th>Overhead</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix (400)</td>
<td>8.5</td>
<td>$1.8 \times 10^3$</td>
</tr>
<tr>
<td>block20 ($10^3$)</td>
<td>1.2</td>
<td>$1.5 \times 10^3$</td>
</tr>
</tbody>
</table>
Matrix Multiplication — Complete Run

- 20x20
- 30x30
- 40x40
- 50x50
- Conventional

Time (s)

Input Size

Conventional

Type-Directed Automatic Incrementalization
Matrix Multiplication — Change Propagation

![Graph showing the time (ms) vs. input size for different matrix sizes. The graph includes data for 50x50, 40x40, 30x30, and 20x20 matrices. The time increases with input size for all matrix sizes.]

Yan Chen
Type-Directed Automatic Incrementalization

22
Matrix Multiplication — Speedup

The diagram shows the speedup of matrix multiplication for different input sizes. The x-axis represents the input size, and the y-axis represents the speedup. The graph includes several lines for different input sizes: 20x20, 30x30, 40x40, and 50x50. The speedup increases as the input size increases, indicating an improvement in performance with larger input sizes.
Ray Tracer

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

```haskell
datatype Surface = T of 
  {ambient: Vector.t C, 
   diffuse: Vector.t C, 
   transmit : real C,...}

type background = Vector.t C
```

Not stable: small input change → large output change
Code “diff”

```kotlin
fun refractRay (dir, normDir, refIndex) =
  let
  val refrIndex = refIndex
  val dotp = -(Vector.dot (dir, normDir))
  val normDir = Vector.normalize (normDir)
  if (dotp < 0.0)
    then (Vector.scale (normDir, -1.0), -dotp, 1.0/refIndex)
  else (normDir, dotp, refIndex) (* trans. only with air *)

val disc = 1.0 - norm^2(1.0-4-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, refDir, intens, contrib) =
  let

  val contrib' = Vector.mul (intens, contrib)

  in

  if Vector.isZero contrib'
    then (0.0, 0.0, 0.0) (* cutoff *)
  else case refDir of
        NONE => (0.0, 0.0, 0.0)
        SOME refDir =>
          let
            (* need to offset just a bit *)
            val ray' = (pos = Vector.add (pos, Vector.scale (refDir, epsilon)),
                        dir = refDir)
            val color' = traceAndShade (ray', contrib')

            in
            Vector.mul (color', intens)
          end

  end

end

(* Reflect a ray from an object *)

and reflectColor (pos, refDir, intens, contrib) =
```

```
```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```
Ray Tracer

Reflective surface  Green diffuse surface

<table>
<thead>
<tr>
<th>Change</th>
<th>Output Diff</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>reflective → diffuse</td>
<td>8.43%</td>
<td>4.29</td>
</tr>
<tr>
<td>diffuse → reflective</td>
<td>8.43%</td>
<td>2.44</td>
</tr>
</tbody>
</table>
Ray Tracer

Reflective background

Diffuse background (red)

<table>
<thead>
<tr>
<th>Change</th>
<th>Output Diff</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>reflective → diffuse</td>
<td>57.22%</td>
<td>1.34</td>
</tr>
<tr>
<td>diffuse → reflective</td>
<td>57.22%</td>
<td>0.22</td>
</tr>
</tbody>
</table>
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