Implicit Self-Adjusting Computation for Purely Functional Programs

Yan Chen
Joshua Dunfield Matthew A. Hammer Umut A. Acar

MPI-SWS

September 19, 2011
Input: 3, 5, 8, 2, 10, 4, 9, 1

Output: Max = 10

- Linear scan: $O(n)$
Incremental/Dynamic Problems

Input: 3, 5, 8, 2, 10, 4, 9, 1

Output: Max = 10 9

- Linear scan: $O(n)$
- Priority queue: $O(\log n)$
Incremental changes are ubiquitous and hard.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Static</th>
<th>Incremental/Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max</td>
<td>[folklore 1950s]</td>
<td>[Williams 1964]</td>
</tr>
<tr>
<td></td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Graph Connectivity</td>
<td>[Strassen 1969]</td>
<td>[Thorup 2000]</td>
</tr>
<tr>
<td></td>
<td>$O(n^{2.8})$</td>
<td>$O(\log n(\log \log n)^3)$ for edge updates</td>
</tr>
<tr>
<td>Planar Convex Hull</td>
<td>[Graham 1972]</td>
<td>[Brodal et al. 2002]</td>
</tr>
<tr>
<td></td>
<td>$O(n \log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Compilation: Whole-program vs. Separate
How can we incrementalize a static algorithm?

```
fun sumOfSquares (x, y) =
  let
    val x2 = x * x
    val y2 = y * y
  in
    x2 + y2
  end
```
How can we incrementalize a static algorithm?

Static Algorithms \(\rightarrow\) 10+ years of research \(\rightarrow\) Incremental Algorithms

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

Dependency Graph
How can we incrementalize a static algorithm?

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

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

Dependency Graph
How can we incrementalize a static algorithm?

Static Algorithms 10+ years of research Incremental Algorithms

fun sumOfSquares (x, y) = 
  let
    val x2 = x * x
    val y2 = y * y
  in
    x2 + y2
  end
How can we incrementalize a static algorithm?

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

Dependency Graph

10+ years of research
Rewrite program to construct dependency graph

fun sumOfSquares (x:int, y:int) = 
  let
    val x2 = x * x
    val y2 = y * y
  in
    x2 + y2
  end
Explicit Self-Adjusting Computation

Rewrite program to construct dependency graph

```
fun sumOfSquares (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.

```
fun sumOfSquares (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
```
Challenges of Explicit Self-Adjusting Computation

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

```ocaml
fun sumOfSquares (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.

```haskell
fun sumOfSquares (x:int mod, y:int) = 
  let
    val x2 = mod (read x as x' in
      write (x' * x' + y * y))
  in
    x2
  end
```
The explicit library is not a natural way of programming.
Efficiency is highly sensitive to program details.
Different requirements lead to different functions.

```haskell
fun sumOfSquares (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
```
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.

```haskell
fun sumOfSquares (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
```
ML Code

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

Explicit Self-Adjusting Code

fun sumOfSquares (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

- Changeable
- Stable
Our Approach

New! Implicit Self-Adjusting Computation

- Annotate input types — no code modification required.
- Automatically infer dependencies from type annotations.
- Polymorphism enables different versions of code.
- Type-directed translation produces an efficient self-adjusting program.
Our Approach

New! Implicit Self-Adjusting Computation

- Annotate input types — no code modification required.
- Automatically infer dependencies from type annotations.
- Polymorphism enables different versions of code.
- Type-directed translation produces an efficient self-adjusting program.
Pure λ-calculus with level annotations.
Use level $C$ to mark changeable data.

Levels $\delta ::= S \mid C \mid \alpha$

Types $\tau ::= \text{int}^\delta \mid (\tau_1 \times \tau_2)^\delta \mid (\tau_1 + \tau_2)^\delta \mid (\tau_1 \rightarrow \tau_2)^\delta$

val sumOfSquares: \text{int}^{\alpha_1} * \text{int}^{\alpha_2} \rightarrow \text{int}^{\alpha_3}$
Pure $\lambda$-calculus with level annotations.

Use level $C$ to mark changeable data.

Levels $\delta ::= S \mid C \mid \alpha$

Types $\tau ::= \text{int}^\delta \mid (\tau_1 \times \tau_2)^\delta \mid (\tau_1 + \tau_2)^\delta \mid (\tau_1 \rightarrow \tau_2)^\delta$

```plaintext
val sumOfSquares: \text{int}^{\alpha_1} \times \text{int}^{\alpha_2} \rightarrow \text{int}^{\alpha_3}
```

Represents:

```plaintext
val sumOfSquaresSS: \text{int}^S \times \text{int}^S \rightarrow \text{int}^S
val sumOfSquaresSC: \text{int}^S \times \text{int}^C \rightarrow \text{int}^C
val sumOfSquaresCS: \text{int}^C \times \text{int}^S \rightarrow \text{int}^C
val sumOfSquaresCC: \text{int}^C \times \text{int}^C \rightarrow \text{int}^C
```
Overview

ML code with level annotations → Type Inference → Translation → Self-Adjusting Program
Identify **affected** computation

```ocaml
fun sumOfSquares (x : int<sup>C</sup>, y : int<sup>S</sup>) =
  let
    val x2 = x * x
    val y2 = y * y
  in
    x2 + y2
  end
```
Type System

- Identify **affected** computation
  - Any data that depends on changeable data must be changeable.

```ocaml
fun sumOfSquares (x:int^C, y:int^S) =
  let
  | val x2 = x * x
  | val y2 = y * y
  in
  x2 + y2
end
```
Identify **affected** computation
- Any data that depends on changeable data must be changeable.

Identify **reusable** computation:

\[
\text{fun sumOfSquares (} x : \text{int}^C, \quad y : \text{int}^S) = \\
\text{let} \\
\qquad \text{val } x^2 = x \times x \\
\qquad \text{val } y^2 = y \times y \\
\text{in} \\
\qquad x^2 + y^2 \\
\text{end}
\]
Identify **affected** computation
- Any data that depends on changeable data must be changeable.

Identify **reusable** computation:
- Non-interference property

```ocaml
fun sumOfSquares (\(x: \mathbb{int}^C\), \(y: \mathbb{int}^S\)) =
  let
    val y2 = y * y
  in
  x2 + y2
end
```
• Identify **affected** computation
  • Any data that depends on changeable data must be changeable.
• Identify **reusable** computation:
  • Non-interference property

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

**Information flow!**
Infer types for all subterms

fun sumOfSquares (x : int$^C$, y : int$^S$) : int =
let
  val x2 : int = x $\times$ x
  val y2 : int = y $\times$ y
  val res : int = x2 + y2
in
  res
end
Infer types for all subterms

```ml
fun sumOfSquares (\(x : \text{int}^C\), \(y : \text{int}^S\)) : \text{int} =
  let
    val \(x^2 : \text{int}^C\) = \(x \times x\)
    val \(y^2 : \text{int}\) = \(y \times y\)
    val \(\text{res} : \text{int}\) = \(x^2 + y^2\)
  in
    \(\text{res}\)
  end
```
Infer types for all subterms

```ocaml
fun sumOfSquares (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
```
Infer types for all subterms

fun sumOfSquares \((x: \text{int}^C, y: \text{int}^S)\) : \text{int} =
let
val \(x2: \text{int}^C\) = \(x \times x\)
val \(y2: \text{int}^S\) = \(y \times y\)
val \(\text{res}: \text{int}^C\) = \(x2 + y2\)
in
res
end
Infer types for all subterms

```ocaml
cfun sumOfSquares (x : int^c, y : int^S) : int^c =
let
  val x2 : int^c = x * x
  val y2 : int^S = y * y
  val res : int^c = x2 + y2
in
  res
end
```
Generate fresh level variables

\[
\alpha \cap \text{FV}(C, \Gamma) = \emptyset
\]

\[
C \land \exists \alpha. D; \Gamma \vdash_\epsilon \text{let } x = v_1 \text{ in } e_2 : \tau
\]

Subsumption

\[
C \vdash \tau' <: \tau''
\]

(SLetV)

Value Restriction

\[
\text{val sumOfSquares: } \text{int}^{\alpha_1} * \text{int}^{\alpha_2} \to \text{int}^{\alpha_3}
\]

\[
[\alpha_3 \geq \alpha_1 \land \alpha_3 \geq \alpha_2]
\]

Our typing rules and constraints fall within the HM(X) framework [Odersky et al. 1999], permitting inference of principal types via constraint solving.
Overview

ML code with level annotations → Type Inference → Translation → Self-Adjusting Program
Modal type system

\( e^C \) has no return value, and can only end with `write` or changeable function application.

Types

\[
\tau ::= \tau \text{ mod} \mid \cdots
\]

Expressions

\[
e ::= e^S \mid e^C
\]

Stable expressions

\[
e^S ::= \text{let } x = e^S \text{ in } e^S
\]

Create

\[
\mid \text{ mod } e^C
\]

\[
\mid \cdots
\]

Changeable expressions

\[
e^C ::= \text{let } x = e^S \text{ in } e^C
\]

Dereference

\[
\mid \text{ read } x \text{ as } y \text{ in } e^C
\]

Store

\[
\mid \text{ write}(x)
\]

\[
\mid \cdots
\]
Overview

ML code with level annotations → Type Inference → Translation → Self-Adjusting Program

Yan Chen  Implicit Self-Adjusting Computation  16
Translation

Source expression \[ \Gamma \vdash e : \tau \mapsto e^\delta \]

Target expression
Translation

Source expression \( \Gamma \vdash e : \tau \xrightarrow{\delta} e^\delta \)

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

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

Yan Chen
Implicit Self-Adjusting Computation

17
Translation

Source expression \( \Gamma \vdash e : \tau \) \( \delta \rightarrow \) Target expression \( e^\delta \)

\[
\begin{align*}
\text{fun} \; \text{sumOfSquares} \; (x, y) &= \text{let} \; \begin{align*}
\text{val} \; x2 &= x * x \\
\text{val} \; y2 &= y * y
\end{align*} \; \text{in} \; \begin{align*}
x2 + y2
\end{align*} \; \text{end}
\end{align*}
\]

\[
\begin{align*}
\text{fun} \; \text{sumOfSquares} \; (x, y) &= \text{let} \; \begin{align*}
\text{val} \; x2 &= \text{mod} (\text{read} \; x \; \text{as} \; x' \; \text{in} \; \text{write} (x' * x')) \\
\text{val} \; y2 &= y * y
\end{align*} \; \text{in} \; \begin{align*}
\text{read} \; x2 \; \text{as} \; x2' \; \text{in} \; \text{write} (x2' + y2) \; \text{end}
\end{align*}
\]

Implicit Self-Adjusting Computation
fun sumOfSquares (x, y) =
  let
    val x2 = x * x
    val y2 = y * y
  in
    x2 + y2
  end

fun sumOfSquares (x, y) =
  let
    val x2 = mod (read x as x' in write (x' * x'))
    val y2 = y * y
  in
    read x2 as x2' in write (x2' + y2)
  end
Typing Environment $\Gamma$: $x_2: \text{int}^C$, $y_2: \text{int}^S$

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

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

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

$$\text{val res} = \overset{s}{\leftarrow} y2$$
Translation Example

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

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

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

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

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

$$\text{val res} = \text{read } x2 \text{ as } x2' \text{ in write } (x2' + y2)$$
Typing Environment $\Gamma$: $x2:\text{int}^C$, $y2:\text{int}^S$, $\text{res}\!:\!\text{int}^C$

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

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

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

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

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

$$\text{s in res}$$
Translation — Monomorphization

\[ \Gamma, x : \forall \vec{\alpha}[D]. \tau' \vdash e : \tau \xrightarrow{\delta} e' \]

\[ \Gamma \vdash v : [\vec{\delta}/\vec{\alpha}]\tau' \xrightarrow{s} e'_i \]

\[ \Gamma \vdash \text{let } x = v \text{ in } e : \tau \xrightarrow{\delta} \text{ let } \{ x_{\delta_i} = e'_i \}_i \text{ in } e' \] (LetV)

\[ \text{val sumOfSquares: } \text{int}^{\alpha_1} \times \text{int}^{\alpha_2} \rightarrow \text{int}^{\alpha_3} \]

\[ [\alpha_3 \geq \alpha_1 \land \alpha_3 \geq \alpha_2] \]

\[ \text{val sumOfSquaresSS: } \text{int}^S \times \text{int}^S \rightarrow \text{int}^S \]

\[ \text{val sumOfSquaresSC: } \text{int}^S \times \text{int}^C \rightarrow \text{int}^C \]

\[ \text{val sumOfSquaresCS: } \text{int}^C \times \text{int}^S \rightarrow \text{int}^C \]

\[ \text{val sumOfSquaresCC: } \text{int}^C \times \text{int}^C \rightarrow \text{int}^C \]

- Dead-code elimination can remove unused functions.
- The functions that are used would have to be handwritten in an explicit setting.
Theoretical Results

ML

$e \xrightarrow{\text{Type inference}} e : \tau \xrightarrow{\text{Evaluation in } k \text{ steps}} v$

Type-directed Translation

Type Soundness

Observational Equivalence

Explicit SAC

$e^\delta : \tau' \xrightarrow{\text{Evaluation in } \Theta(k) \text{ steps}} w$
Summary

- **Implicit** Self-Adjusting Computation
  - Automatic dependency tracking based on type annotation
  - Type-directed translation for self-adjusting computation

- **Automatically** make ML programs self-adjusting
  - Formal proofs of translation soundness and asymptotic complexity
  - Implementation and preliminary results presented at Workshop on ML

See paper!