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: $O(n)$

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

Output: Max $=$ ~~10~~ 9

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

Motivation

Incremental changes are ubiquitous and hard.

Problem	Static	Incremental/Dynamic
Max	[folklore 1950s] $O(n)$	[Williams 1964] $O(\log n)$
Graph Connectivity	[Strassen 1969] $O(n^{2.8})$	[Thorup 2000] $O\left(\log n(\log \log n)^3\right)$ for edge updates
Planar Convex Hull	[Graham 1972] $O(n \log n)$	[Brodal et al. 2002] $O(\log n)$
⋮		
Compilation	Whole-program	Separate

Challenge

How can we incrementalize a static algorithm?

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

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

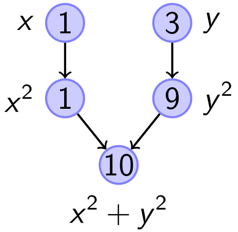
Challenge

How can we incrementalize a static algorithm?

Static Algorithms → Incremental Algorithms

10+ years of research

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



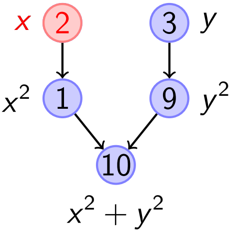
Dependency Graph

Challenge

How can we incrementalize a static algorithm?

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

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



Dependency Graph

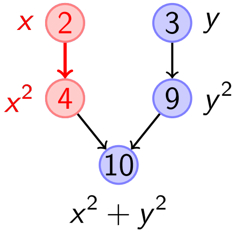
Challenge

How can we incrementalize a static algorithm?

Static Algorithms $\longrightarrow$ Incremental Algorithms

10+ years of research

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



Dependency Graph

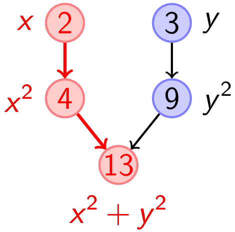
Challenge

How can we incrementalize a static algorithm?

Static Algorithms → Incremental Algorithms

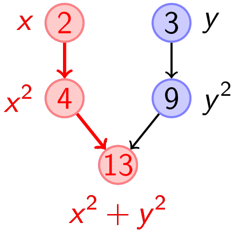
10+ years of research

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



Dependency Graph

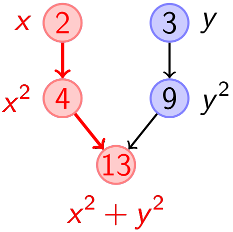
Rewrite program to construct dependency graph




```
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.

```
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.

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

  in

    x2
  end
```

Challenges 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 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
```

Challenges 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 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
```

Bridge the Gap

ML Code

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

$\hookrightarrow$ **?**

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
```

x **Changeable**

y **Stable**

Our Approach

New! **Implicit** Self-Adjusting Computation

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

- 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.

New! **Implicit** Self-Adjusting Computation

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

- 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.

Source Language

- Pure λ-calculus with level annotations.
- Use level $\mathcal{C}$ to mark changeable data.

$$\textit{Levels}\ \delta ::= \mathcal{S} \mid \mathcal{C} \mid \alpha$$
$$\textit{Types}\quad \tau ::= \textbf{int}^{\delta} \mid (\tau_1 \times \tau_2)^{\delta} \mid (\tau_1 + \tau_2)^{\delta} \mid (\tau_1 \rightarrow \tau_2)^{\delta}$$

$\texttt{val sumOfSquares:}\quad \texttt{int}^{\alpha_1}\ \texttt{*}\ \texttt{int}^{\alpha_2}\ \texttt{->}\ \texttt{int}^{\alpha_3}$

Source Language

- Pure λ-calculus with level annotations.
- Use level $\mathcal{C}$ to mark changeable data.

$$\textit{Levels}\ \delta ::= \mathcal{S} \mid \mathcal{C} \mid \alpha$$
$$\textit{Types}\ \ \tau ::= \mathbf{int}^{\delta} \mid (\tau_1 \times \tau_2)^{\delta} \mid (\tau_1 + \tau_2)^{\delta} \mid (\tau_1 \rightarrow \tau_2)^{\delta}$$

$$\texttt{val sumOfSquares: int}^{\alpha_1}\ \texttt{* int}^{\alpha_2}\ \texttt{-> int}^{\alpha_3}$$

Represents:

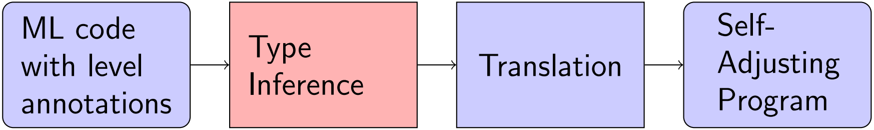
$\texttt{val sumOfSquaresSS: int}^{\mathcal{S}}\ \texttt{* int}^{\mathcal{S}}\ \texttt{-> int}^{\mathcal{S}}$

$\texttt{val sumOfSquaresSC: int}^{\mathcal{S}}\ \texttt{* int}^{\mathcal{C}}\ \texttt{-> int}^{\mathcal{C}}$

$\texttt{val sumOfSquaresCS: int}^{\mathcal{C}}\ \texttt{* int}^{\mathcal{S}}\ \texttt{-> int}^{\mathcal{C}}$

$\texttt{val sumOfSquaresCC: int}^{\mathcal{C}}\ \texttt{* int}^{\mathcal{C}}\ \texttt{-> int}^{\mathcal{C}}$

Overview


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

Type System

- Identify **affected** computation

```
fun sumOfSquares (x:int^C, y:int^S) =
  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.

```
fun sumOfSquares (x:int^C, y:int^S) =
  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.
- Identify **reusable** computation:

```
fun sumOfSquares (x:int^C, y:int^S) =
  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.
- Identify **reusable** computation:
  - Non-interference property

```
fun sumOfSquares ( :int^C, y :int^S) =
  let

    val y2 = y * y
  in

  end
```

Type System

- Identify **affected** computation
  - Any data that depends on changeable data must be changeable.
- Identify **reusable** computation:
  - Non-interference property

```
fun sumOfSquares (   :int^C, y :int^S) =
  let

    val y2 = y * y
  in

  end
```

Information flow!

Type Inference for Concrete Levels

Infer types for all subterms

```
fun sumOfSquares ( 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
```

Type Inference for Concrete Levels

Infer types for all subterms

```
fun sumOfSquares (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
```

Type Inference for Concrete Levels

Infer types for all subterms

```
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
```

Type Inference for Concrete Levels

Infer types for all subterms

```
fun sumOfSquares (x:intᶜ, y:intˢ) : int =
  let
    val x2:intᶜ = x * x
    val y2:intˢ = y * y
    val res:intᶜ = x2 + y2
  in
     res
  end
```

Type Inference for Concrete Levels

Infer types for all subterms

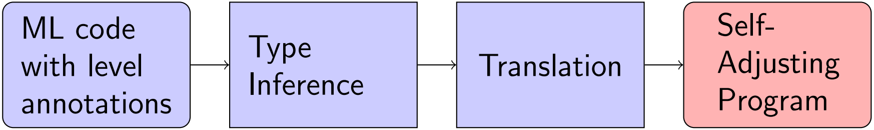
```
fun sumOfSquares (x:int^C, y:int^S) : int^C =
  let
    val x2:int^C = x * x
    val y2:int^S = y * y
    val res:int^C = x2 + y2
  in
    res
  end
```

$$\dfrac{C \wedge D; \Gamma \vdash_{\mathcal{S}} v_1 : \tau' \qquad C; \Gamma, x : \forall\vec{\alpha}[D].\,\tau'' \vdash_{\varepsilon} e_2 : \tau \qquad \overbrace{\vec{\alpha} \cap FV(C, \Gamma) = \emptyset}^{\textbf{Generate fresh level variables}} \qquad \overbrace{C \Vdash \tau' <: \tau''}^{\textbf{Subsumption}}}{C \wedge \exists\vec{\alpha}.D; \Gamma \vdash_{\varepsilon} \underbrace{\textbf{let } x = v_1 \textbf{ in } e_2}_{\textbf{Value Restriction}} : \tau} \text{(SLetV)}$$

val sumOfSquares: $\texttt{int}^{\alpha_1} * \texttt{int}^{\alpha_2} \texttt{ -> } \texttt{int}^{\alpha_3}$
$[\alpha_3 \geq \alpha_1 \wedge \alpha_3 \geq \alpha_2]$

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

Overview

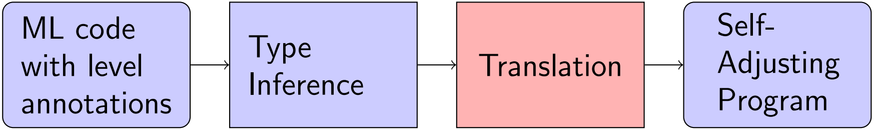

ML code with level annotations
Type Inference
Translation
Self-Adjusting Program

Target Language

- Modal type system
- $e^{\mathcal{C}}$ has no return value, and can only end with **write** or changeable function application.

$$
\begin{array}{llcll}
\textit{Types} & \underline{\tau} & ::= & \underline{\tau}\ \textbf{mod} \mid \cdots & \\
\textit{Expressions} & e & ::= & e^{\mathcal{S}} \mid e^{\mathcal{C}} & \\
\textit{Stable} & e^{\mathcal{S}} & ::= & \textbf{let}\ x = e^{\mathcal{S}}\ \textbf{in}\ e^{\mathcal{S}} & \\
\textit{expressions} & & \mid & \textbf{mod}\ e^{\mathcal{C}} & \textbf{create} \\
 & & \mid & \ldots & \\
\textit{Changeable} & e^{\mathcal{C}} & ::= & \textbf{let}\ x = e^{\mathcal{S}}\ \textbf{in}\ e^{\mathcal{C}} & \\
\textit{expressions} & & \mid & \textbf{read}\ x\ \textbf{as}\ y\ \textbf{in}\ e^{\mathcal{C}} & \textbf{dereference} \\
 & & \mid & \textbf{write}(x) & \textbf{store} \\
 & & \mid & \ldots &
\end{array}
$$

Overview


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

Translation

Source expression Target expression

$$\Gamma \vdash e : \tau \xrightarrow[\delta]{} e^{\delta}$$

Translation

Source expression Target expression

$$\Gamma \vdash e : \tau \xhookrightarrow[\delta]{} e^{\delta}$$

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

$\xhookrightarrow[\mathcal{C}]{}$

```
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
```

Translation

Source expression Target expression

$$\Gamma \vdash e : \tau \xhookrightarrow[\delta]{} e^{\delta}$$

$$\Downarrow \qquad\qquad \Downarrow$$

$$v \qquad\qquad w$$

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

$\xhookrightarrow[\mathcal{C}]{}$

```
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
```

Translation

Source expression Target expression

$$\Gamma \vdash e : \tau \xhookrightarrow[\delta]{} e^{\delta}$$

$$\Downarrow \qquad \Downarrow$$

$$v \equiv w$$

Correctness

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

$\xhookrightarrow[\mathcal{C}]{}$

```
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
```

Translation Example

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

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

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

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
```

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
```

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
                 write (x2' + y2)
```

Translation Example

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

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

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

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

Translation Example

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

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

$\underset{S}{\hookrightarrow}$

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

Translation — Monomorphization

$$\frac{\Gamma, x : \forall\vec{\alpha}[D].\,\tau' \vdash e : \tau \underset{\delta}{\hookrightarrow} e' \qquad \overbrace{\begin{array}{r}\text{For all } \vec{\delta_i} \text{ s.t. } \vec{\alpha} = \vec{\delta_i} \Vdash D,\\ \Gamma \vdash v : [\vec{\delta_i}/\vec{\alpha}]\tau' \underset{\mathcal{S}}{\hookrightarrow} e_i'\end{array}}^{\textbf{Generate all satisfying instances}}}{\Gamma \vdash \textbf{let } x = v \textbf{ in } e \;:\; \tau \;\underset{\delta}{\hookrightarrow}\; \textbf{let } \{x_{\vec{\delta_i}} = e_i'\}_i \textbf{ in } e'} \;(\text{LetV})$$

```
val sumOfSquares:  int^α1 * int^α2 -> int^α3
                   [α3 ≥ α1 ∧ α3 ≥ α2]
```

$\underset{\delta}{\hookrightarrow}$

```
val sumOfSquaresSS: int^S * int^S -> int^S
val sumOfSquaresSC: int^S * int^C -> int^C
val sumOfSquaresCS: int^C * int^S -> int^C
val sumOfSquaresCC: int^C * int^C -> 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

$$
\begin{tikzcd}[column sep=large, row sep=huge]
\text{ML} & e \arrow[r, "\text{Type inference}"] & e : \tau \arrow[r, "\text{Evaluation}", "\text{in } k \text{ steps}"'] \arrow[d, hook, "\text{Type-Directed Translation}"'] \arrow[d, phantom, shift left=3ex, "\text{Type Soundness}"] & v \arrow[d, equal, "\text{Observational Equivalence}"] \\
\text{Explicit SAC} & & e^{\delta} : \tau' \arrow[r, "\text{Evaluation}", "\text{in } \Theta(k) \text{ steps}"'] & w
\end{tikzcd}
$$

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!