Implementing Implicit Self-Adjusting Computation

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

MPI-SWS

ML ’11, September 18, 2011
Incremental/Dynamic Problems

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: $O(\log n)$
How can we incrementalize a static algorithm?

```ml
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
Challenge

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

Yan Chen
Implementing Implicit Self-Adjusting Computation
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?

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

```haskell
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.

```haskell
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
```

Yan Chen
Implementing Implicit 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' + 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
```
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.

```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
```
**Our Approach**

New! **Implicit Self-Adjusting Computation**

**ML Code**

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

\( C \) Changeable

\( S \) Stable

---

Yan Chen | Implementing Implicit Self-Adjusting Computation | 6
Implementation Structure

ML with level types

Type Inference → Constraints → Solver

SXXML → Translation Algorithm

AFL Library + ML code → Optimization
Implementation Structure

ML with level types

Type Inference

Constraints

Solver

Translation Algorithm

Optimization

SXXML

AFL Library

ML code

+
Implementation Structure

- ML with level types
  - Type Inference
    - Constraints
    - Solver
  - SXML
  - Translation Algorithm
  - Optimization
- AFL Library
  + ML code
Implementation Structure

ML with level types

Type Inference

Constraints

Solver

SXML

Translation Algorithm

Optimization

AFL Library

ML code
Implementation Structure

ML with level types

Type Inference

Constraints

Solver

Translation Algorithm

Optimization

SXXML

AFL Library

ML code
Implementation Structure

ML with level types

Type Inference

Constraints

Solver

AFL Library

ML code

SXML

Translation Algorithm

Optimization

Yan Chen
Implementing Implicit Self-Adjusting Computation
SML extended with level types

Specify SML program with $C/S$

datatype array $S = \text{nil}$
  | cons of int $C \times \text{list } S$

datatype list $C = \text{nil}$
  | cons of int $S \times \text{list } C$

type pair = (int $C \times \text{int } C) \times S$

type point = (int $S \times \text{int } S) \times C$
Change Propagation

Time for Change Propagation

- **map**
- **filter**
- **reverse**
- **split**
- **qsort**
- **msort**

- **CPS**
- **Translate**
- **Opt**
- **AFL**

Yan Chen
Implementing Implicit Self-Adjusting Computation
Future Work

- Memoization
  - Cross different program executions
  - Identify memoization points and keys
  - Nondeterministic allocation
- Granularity control
  - Trade-off between time and space
  - Similar idea from parallelism
- Mutable references
Thank you!
For simplicity, the translation algorithm is local.
  It can generate slow code:

\[
y : \text{int} \rightarrow \text{mod} (\text{read } y \text{ as } y' \text{ in write } y')
\]

But this is equivalent to just \( y \).

After running the translation algorithm, we apply rewrite rules to fix things like the above.
Yan Chen
Implementing Implicit Self-Adjusting Computation