Compiling Self-Adjusting Programs with Continuations

Ruy Ley-Wild[1] Matthew Fluet[2] Umut Acar[2]

[1]Carnegie Mellon University

[2]Toyota Technological Institute at Chicago

September 24, 2008

Dealin with Input Changes

Dealin with Input Changes

```
  \end{center}
\end{frame}


\begin{frame}
  \frametitle{Dealin with Input Changes}

\end{frame}


\begin{frame}
  \frametitle{Self-Adjusting Computation}
```

talk.tex

Dealin with Input Changes

```
  \end{center}
\end{frame}


\begin{frame}
  \frametitle{Dealin with Input Changes}

\end{frame}


\begin{frame}
  \frametitle{Self-Adjusting Computation}
```

`talk.tex`

- First run **must** process the entire source file

Dealin with Input Changes

`talk.tex`

- First run **must** process the entire source file

Dealin with Input Changes

```
  \end{center}
\end{frame}


\begin{frame}
  \frametitle{Dealing with Input Changes}

\end{frame}


\begin{frame}
  \frametitle{Self-Adjusting Computation}
```

`talk.tex`

- First run **must** process the entire source file

Dealing with Input Changes

```
  \end{center}
\end{frame}


\begin{frame}
  \frametitle{Dealing with Input Changes}

\end{frame}


\begin{frame}
  \frametitle{Self-Adjusting Computation}
```

`talk.tex`

- First run **must** process the entire source file

Dealing with Input Changes

```
  \end{center}
\end{frame}


\begin{frame}
  \frametitle{Dealing with Input Changes}

\end{frame}


\begin{frame}
  \frametitle{Self-Adjusting Computation}
```

`talk.tex`

- First run **must** process the entire source file
- Second run **only** has to process the affected frame

  ...but `latex` processes the entire file again

Self-Adjusting Computation

Self-Adjusting Computation (SAC)
= **Recompute** affected outputs + **Reuse** unaffected outputs

- Modes of execution
  - From scratch
  - Update: reuse work from previous runs
- Track computation that depends on input changes

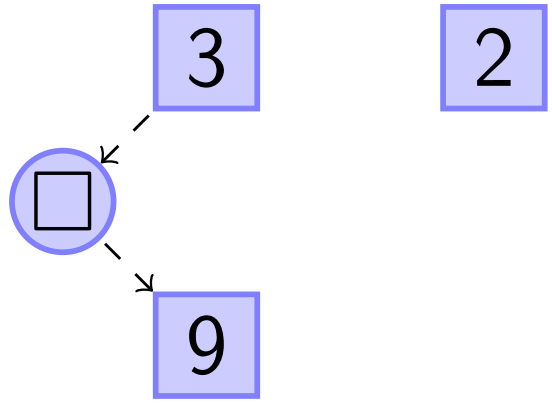
Example: Sum of Squares

from scratch run: sumOfSquares(3,2)

data

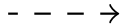
Example: Sum of Squares

from scratch run: sumOfSquares(3,2)



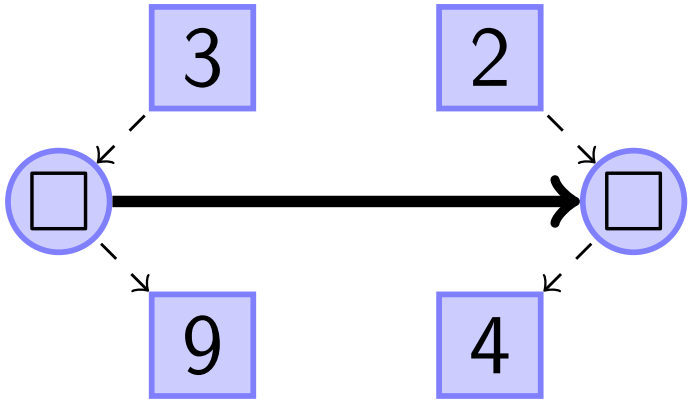

data


computation


data flow

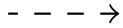
Example: Sum of Squares

from scratch run: sumOfSquares(3,2)




data

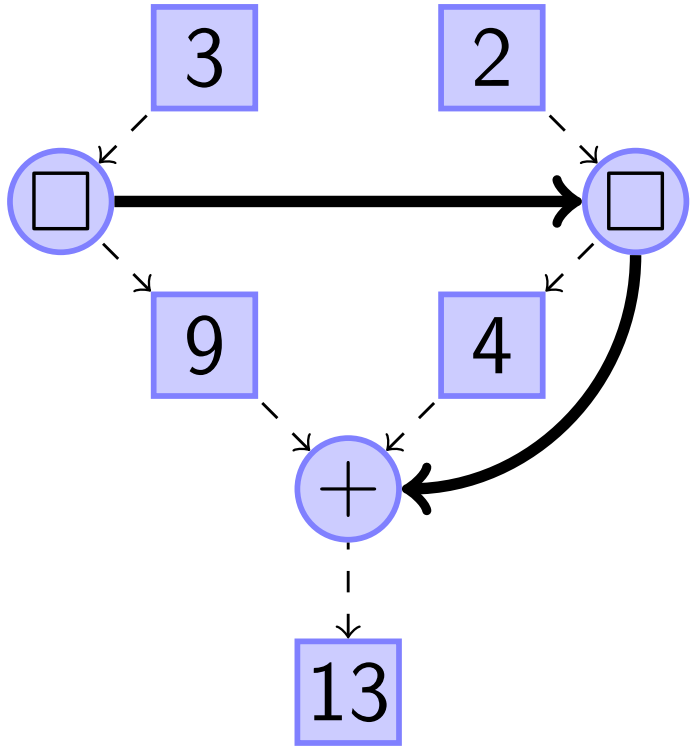

computation

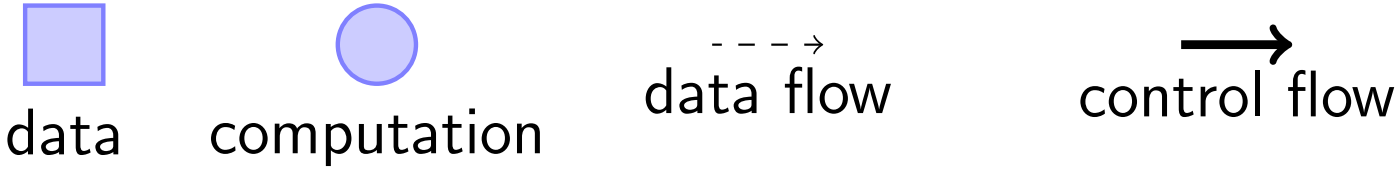

data flow


control flow

Example: Sum of Squares

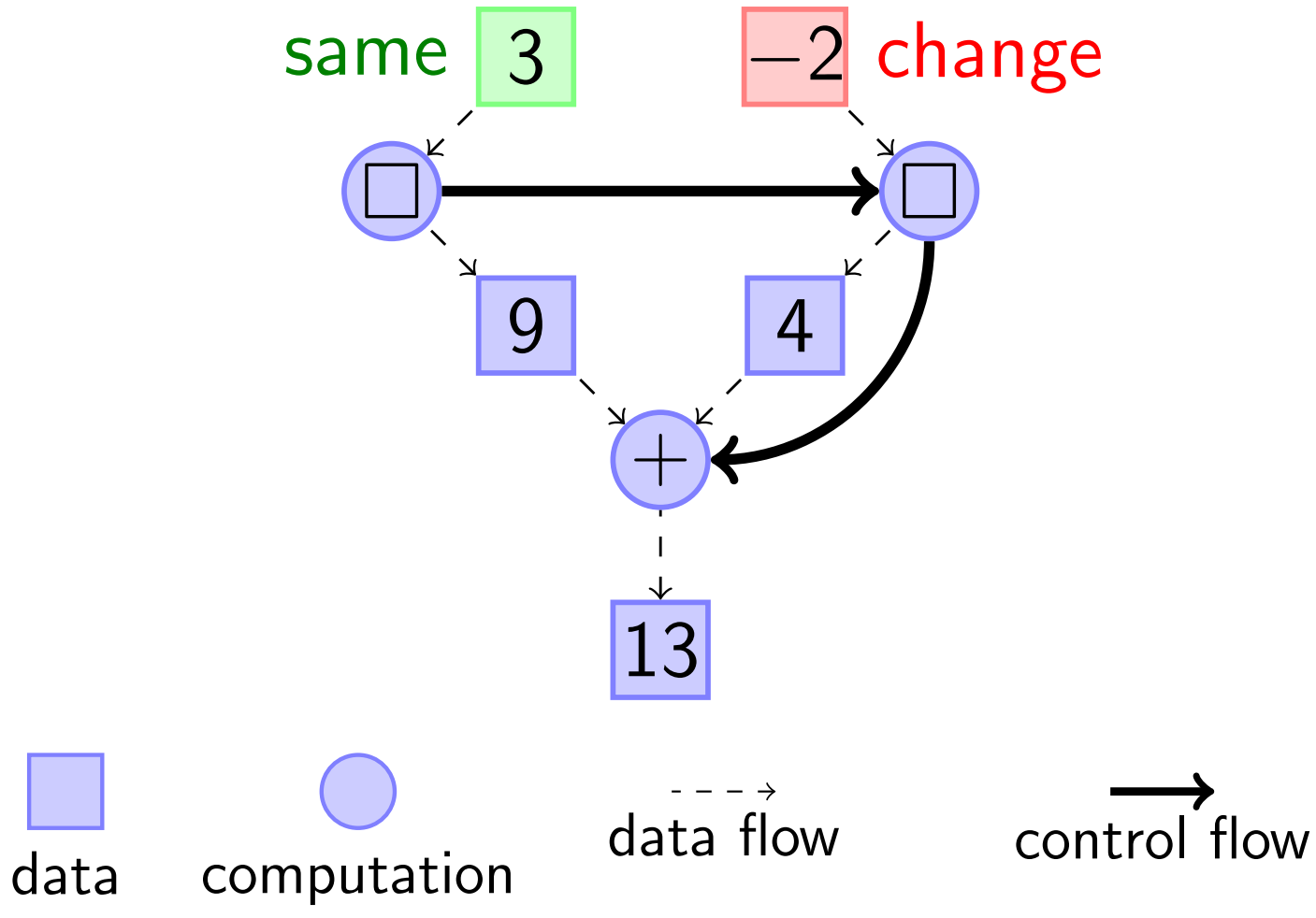
from scratch run: sumOfSquares(3,2)





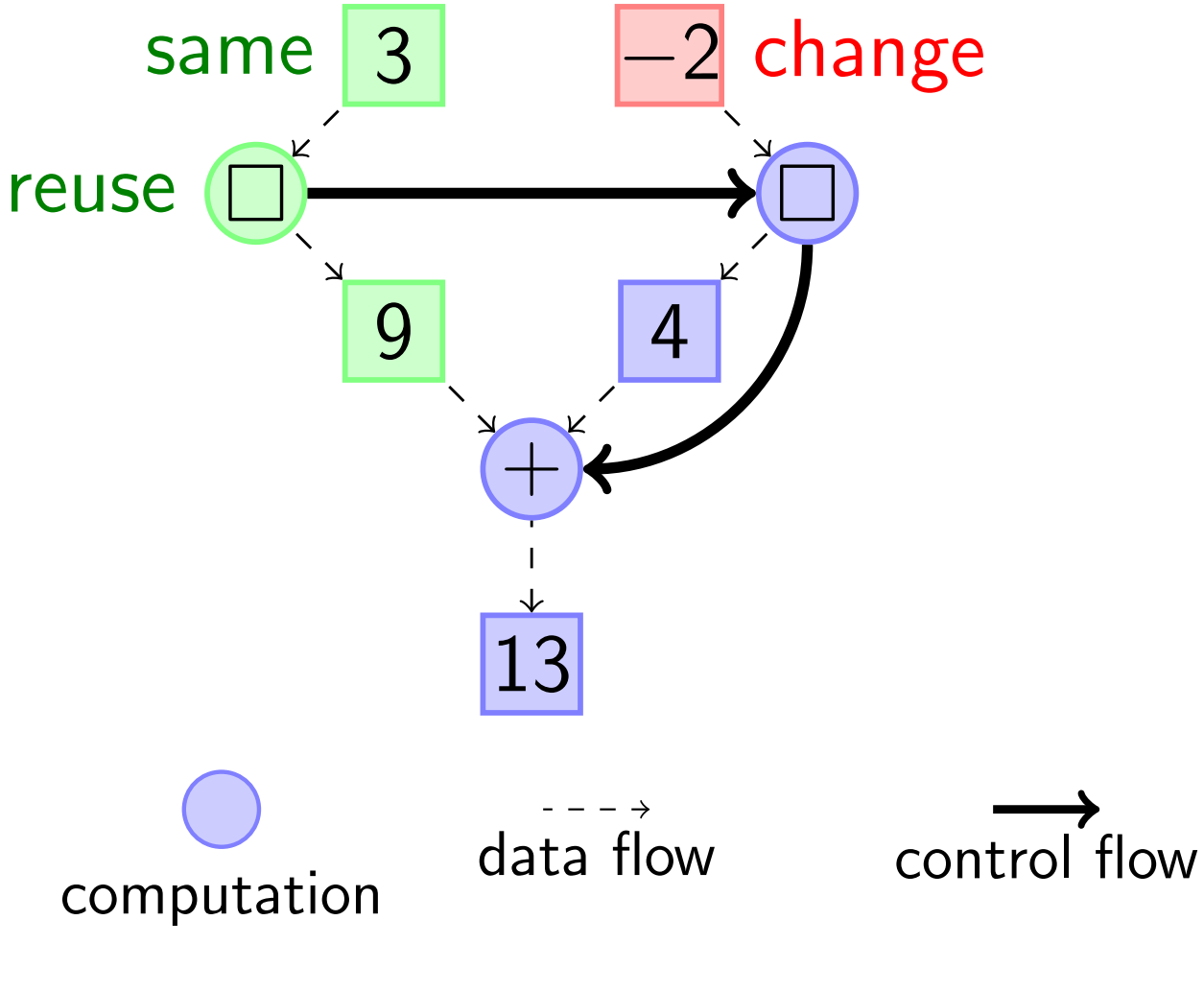
Example: Sum of Squares

from scratch run: sumOfSquares(3,2)
update run: sumOfSquares(3,−2)



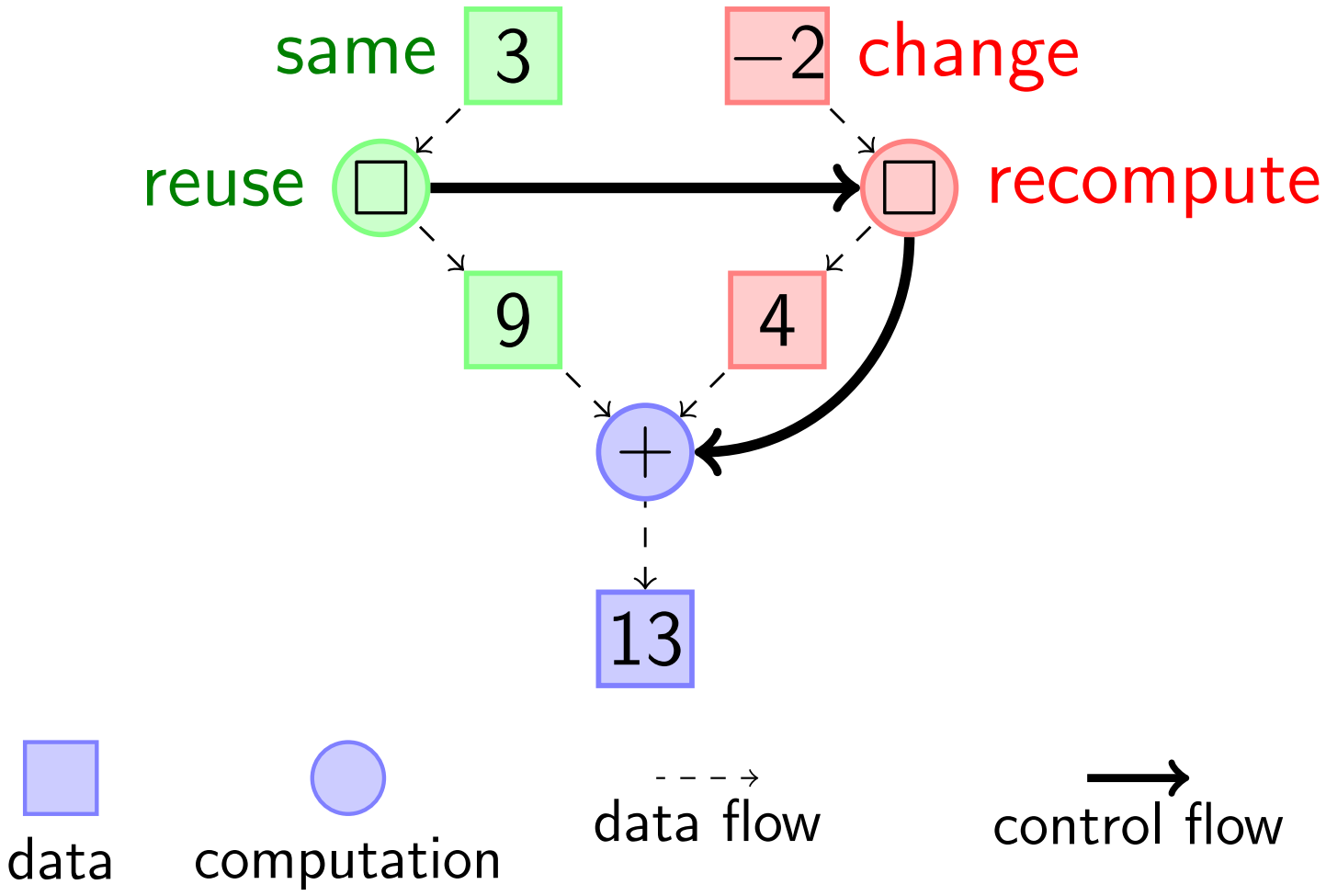
Example: Sum of Squares

from scratch run: sumOfSquares(3,2)
update run: sumOfSquares(3,−2)



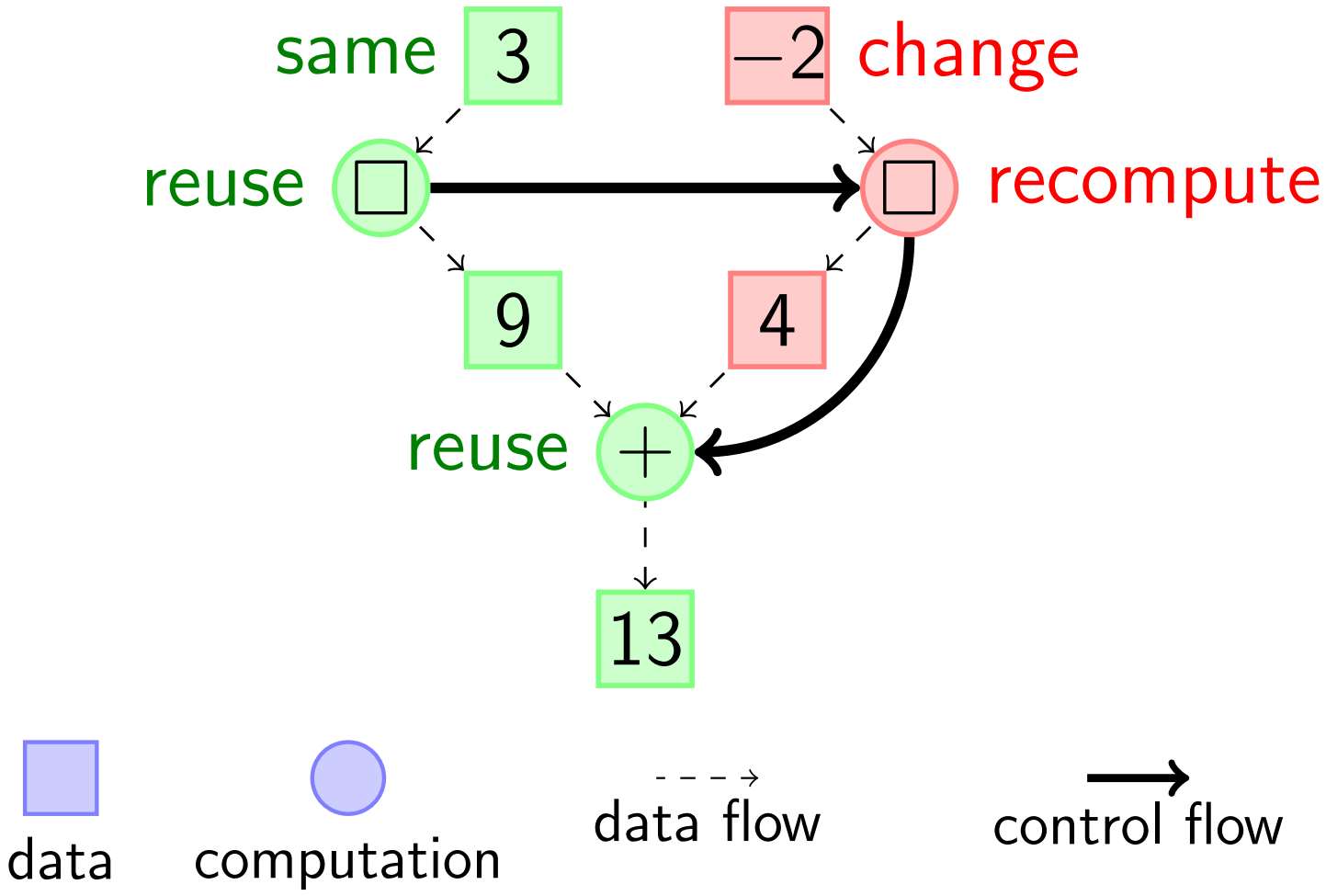
Example: Sum of Squares

from scratch run: sumOfSquares(3,2)
update run: sumOfSquares(3,−2)



Example: Sum of Squares

from scratch run: sumOfSquares(3,2)
update run: sumOfSquares(3,−2)



Applications

- Algorithms (computational geometry, dynamic algorithms)
- Machine learning (Bayesian inference)
- Robotics
- Software verification (dynamic invariant checking)
- Hardware design (reconfigurable hardware)

Tracking Dependencies

Previous approach: **Manually** track dependencies

- monadic destination-passing primitives
- manual hashing, equality, memoization

```
fun sumOfSquares (x, y) =
  let x2 = read(x, fn m => write(m * m))
      y2 = read(y, fn n => write(n * n))
  in memo (x2,y2) (fn (x2,y2) => read(x2, fn n2 =>
       read(y2, fn m2 => write(n2 + m2)))) end
```

Tracking Dependencies

Previous approach: **Manually** track dependencies

- monadic destination-passing primitives
- manual hashing, equality, memoization

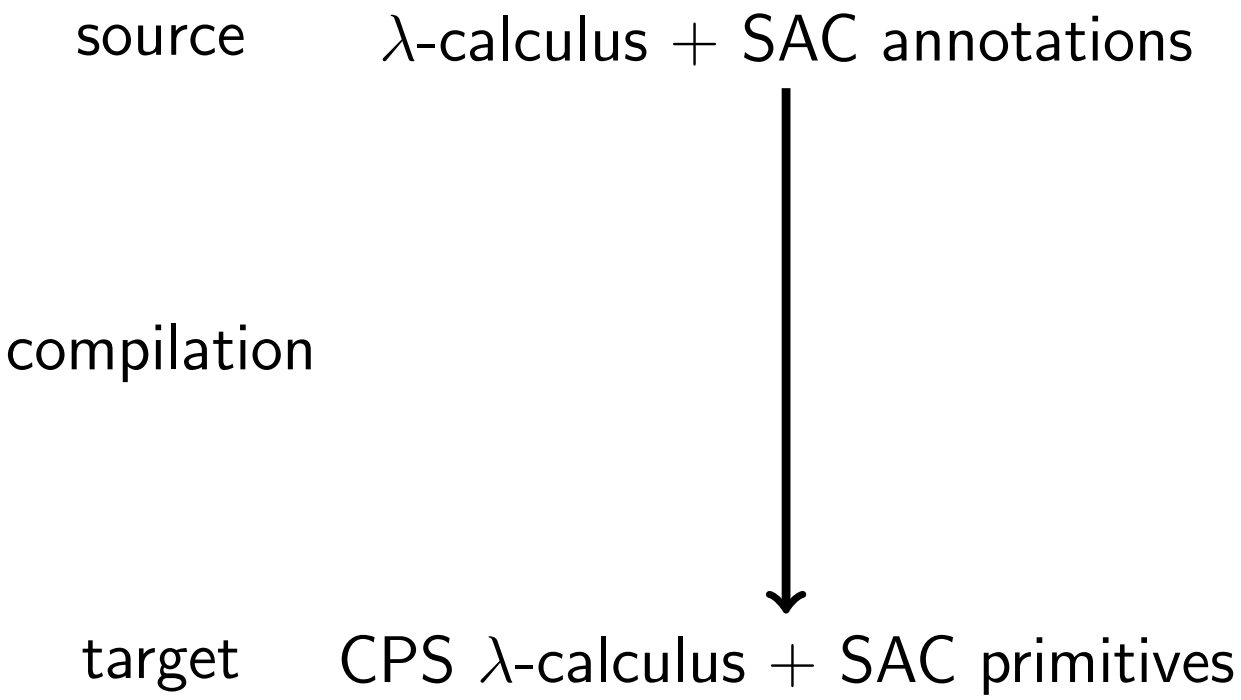
```
fun sumOfSquares (x, y) =
  let x2 = read(x, fn m => write(m * m))
      y2 = read(y, fn n => write(n * n))
  in memo (x2,y2) (fn (x2,y2) => read(x2, fn n2 =>
       read(y2, fn m2 => write(n2 + m2)))) end
```

New approach: **Automatically** infer dependencies

- Language support: direct-style annotations
- Compiler support: infer dependencies from annotations

```
fun sumOfSquares (box m, box n) =
  box(unbox (box(m * m)) + unbox(box (n * n)))
```

Compilation Outline



Source Language

- Pure λ-calculus with SAC **annotations**
- Use **boxes** to mark **changeable** data

$$\tau ::= \tau \ \textbf{box} \mid \cdots$$

$$e ::= \textbf{box}\ e \qquad \textbf{create}$$

$$\mid \textbf{unbox}\ e \qquad \textbf{dereference}$$

$$\mid \cdots$$

Source Language

- Pure λ-calculus with SAC **annotations**
- Use **boxes** to mark **changeable** data

$$\tau ::= \tau \ \textbf{box} \mid \cdots$$

$$e ::= \textbf{box}\ e \qquad \textbf{create}$$

$$\mid \textbf{unbox}\ e \qquad \textbf{dereference}$$

$$\mid \cdots$$

```
datatype 'a list = nil | ::  of 'a * 'a list
map :  ('a -> 'b) -> 'a list     -> 'b list
fun map f (    nil) =     nil
  | map f (    (h::t)) =     (f h ::  map f t)
```

Source Language

- Pure λ-calculus with SAC **annotations**
- Use **boxes** to mark **changeable** data

$$\tau ::= \tau \ \textbf{box} \mid \cdots$$

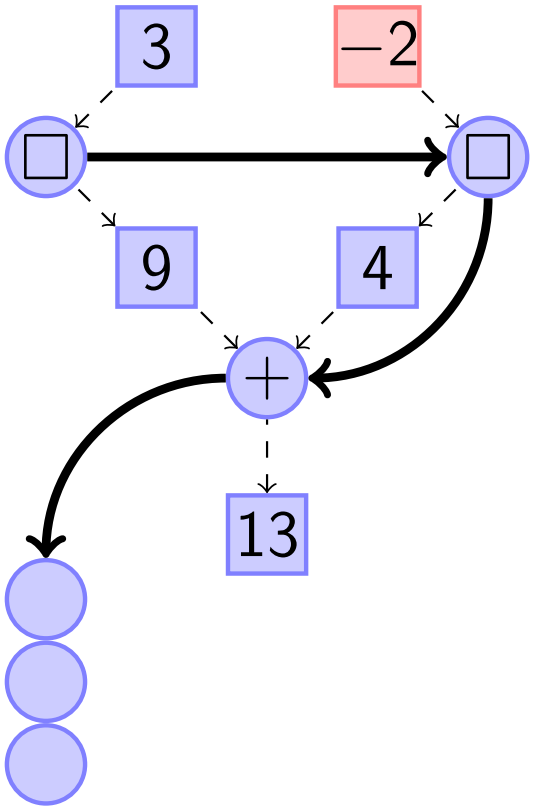
$$e ::= \textbf{box}\ e \qquad \textbf{create}$$

$$\mid \textbf{unbox}\ e \qquad \textbf{dereference}$$

$$\mid \cdots$$

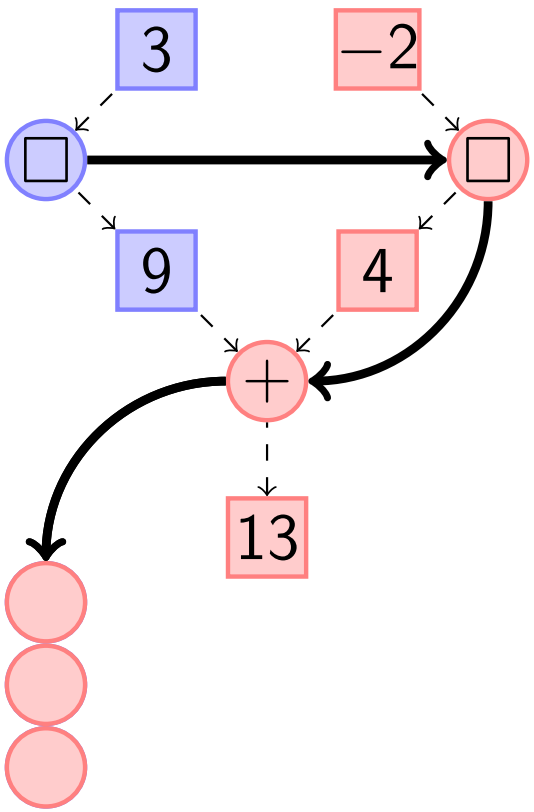
```
datatype 'a list = nil | ::  of 'a * 'a list box
map :  ('a -> 'b) -> 'a list box -> 'b list box
fun map f (box nil) = box nil
  | map f (box (h::t)) = box (f h ::  map f t)
```

Challenges of Compilation



How to identify control dependencies?

Challenges of Compilation

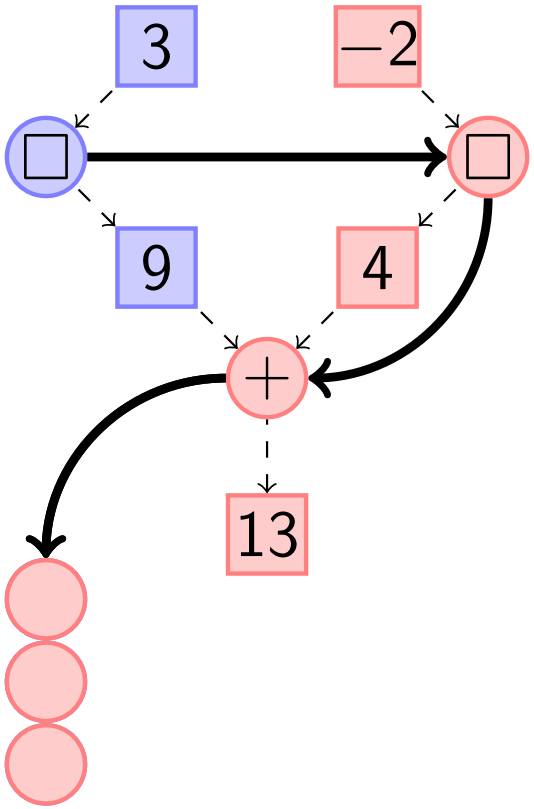


How to identify control dependencies?

Use the **continuation**!

= the rest of the computation

Challenges of Compilation



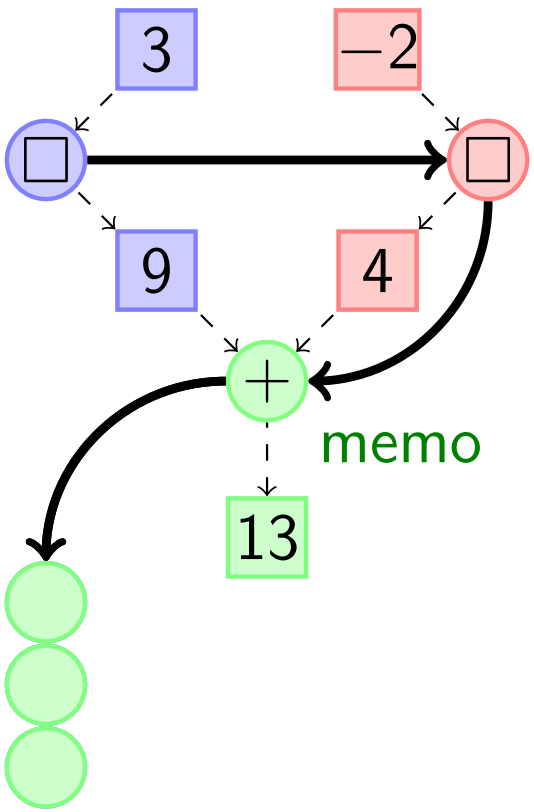
How to identify control dependencies?

Use the **continuation**!

= the rest of the computation

But continuation is an **overapproximation**

Challenges of Compilation



How to identify control dependencies?

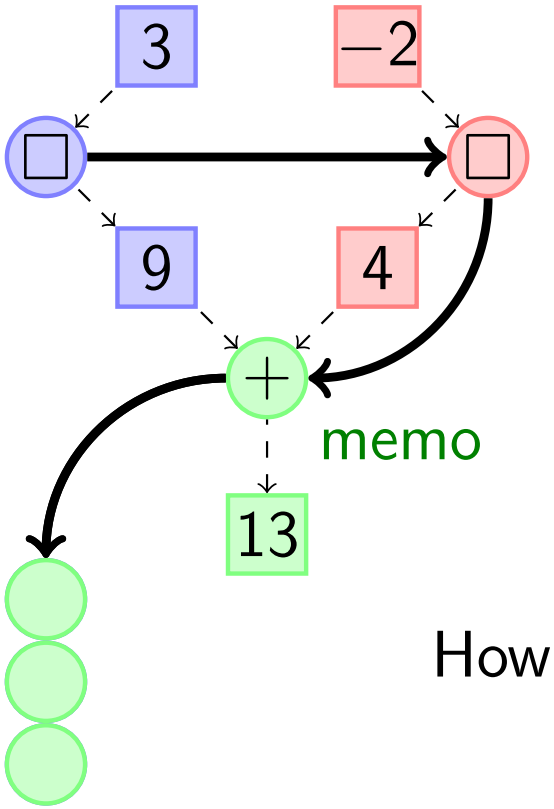
Use the **continuation**!

= the rest of the computation

But continuation is an **overapproximation**

Use **memoization**

Challenges of Compilation



How to identify control dependencies?

Use the **continuation**!

= the rest of the computation

But continuation is an **overapproximation**

Use **memoization**

How to **combine** memoization and continuations?

Target Language

- Pure CPS λ-calculus with SAC **primitives**

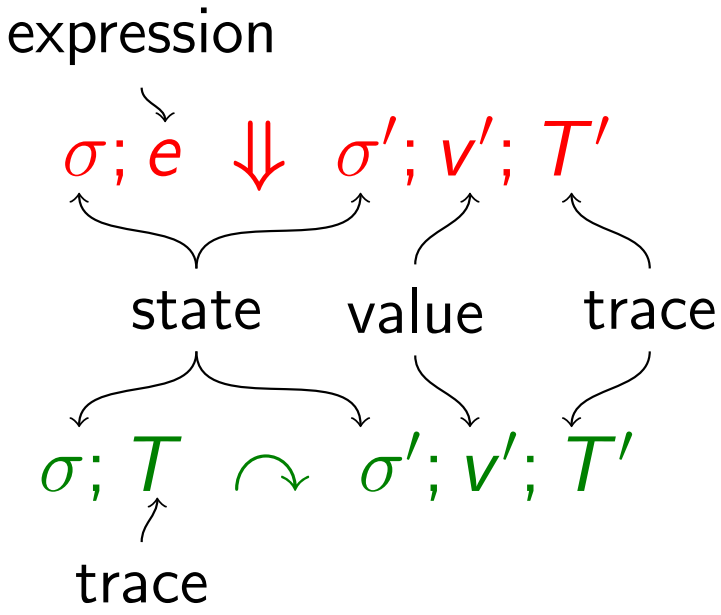
$$
\begin{aligned}
e ::= \; & \textbf{fun}\, f.x.k.e && \textbf{explicit continuation} \\
\mid \; & v_f \; v_x \; v_k \\
\mid \; & \textbf{boxk}\; v \; v_k \\
\mid \; & \textbf{unboxk}\; v \; v_k \\
\mid \; & \textbf{memo}\; e && \textbf{attempt reuse} \\
\mid \; & \ldots
\end{aligned}
$$

- Intrinsic support for self-adjusting computation

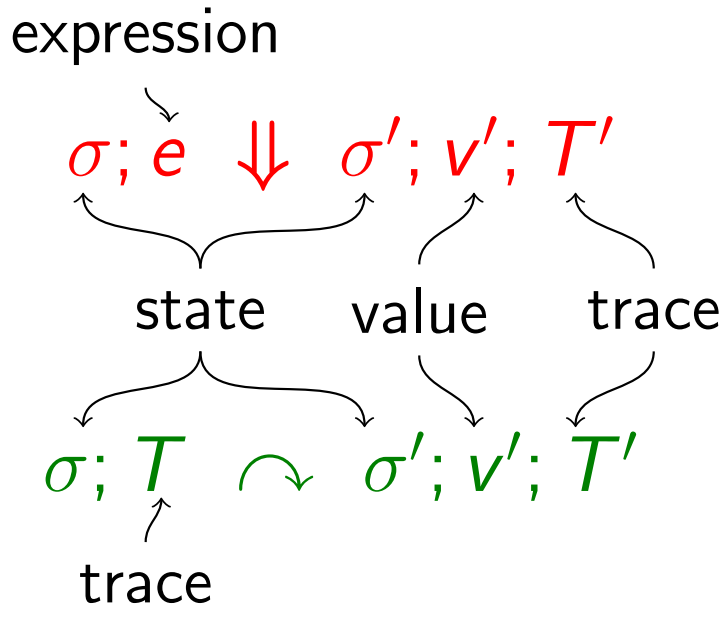
Dynamic Semantics for **Evaluation** and **Replay**

evaluation

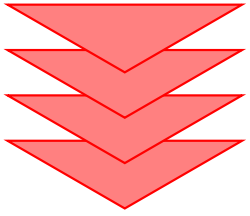
replay



Dynamic Semantics for **Evaluation** and **Replay**

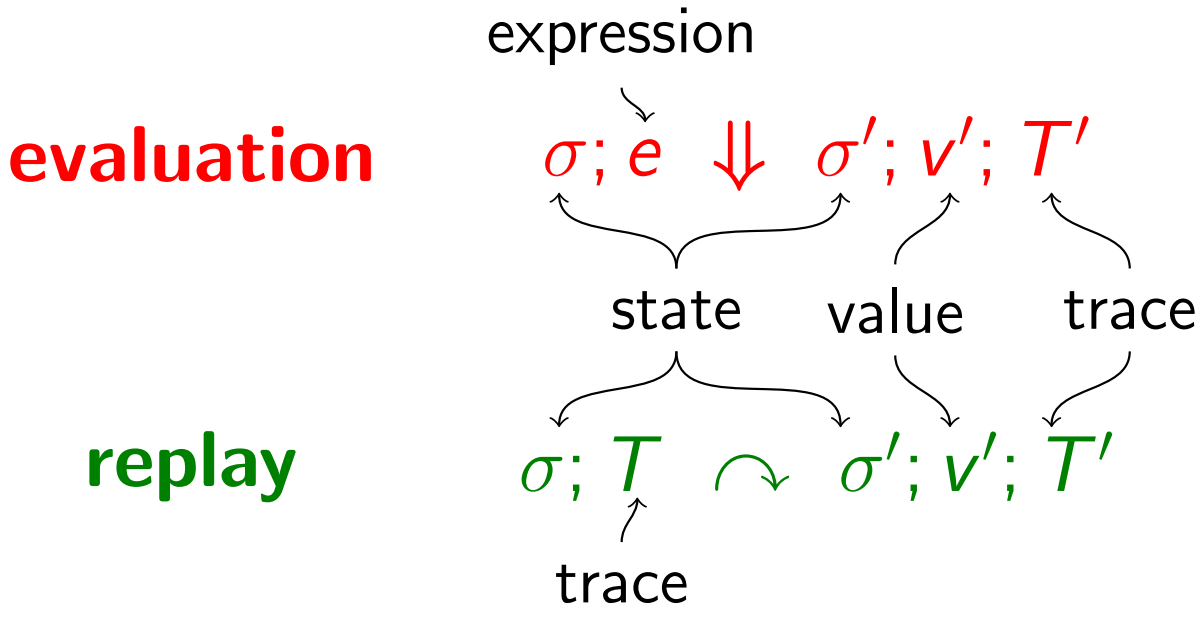


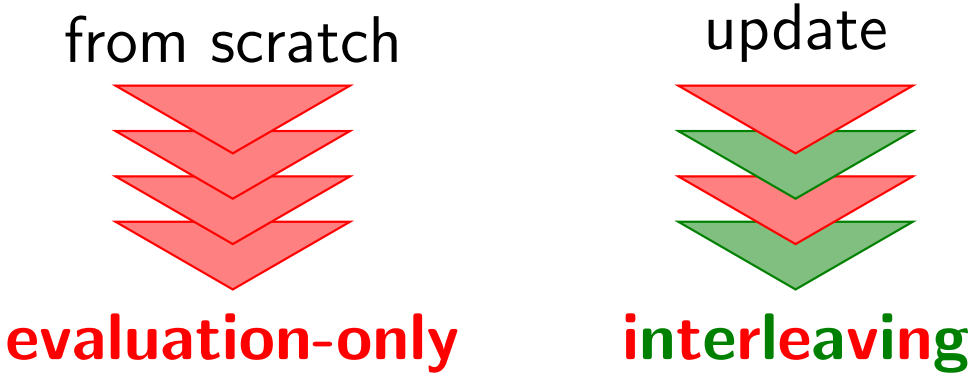
from scratch



evaluation-only

Dynamic Semantics for **Evaluation** and **Replay**

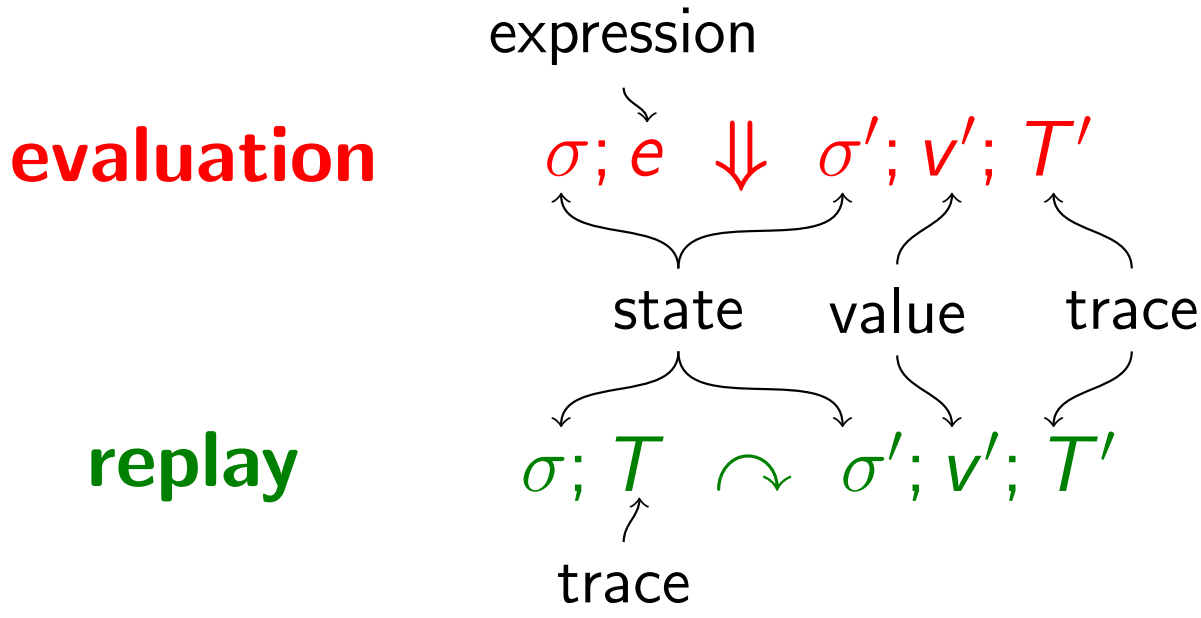




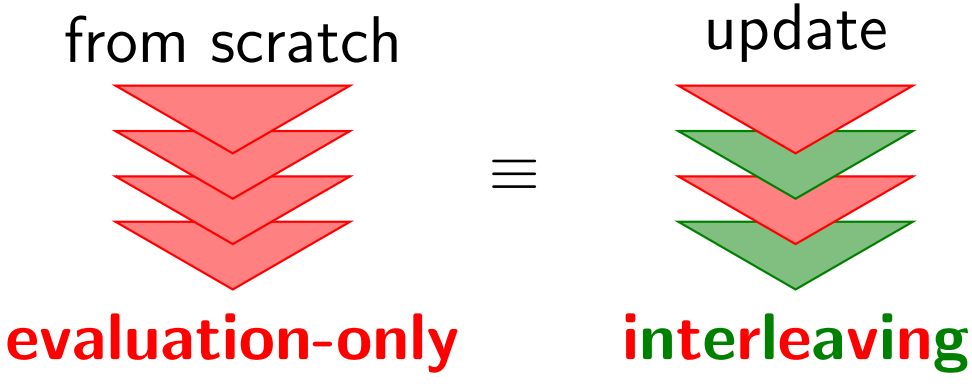
replay when possible

evaluate when necessary

Dynamic Semantics for **Evaluation** and **Replay**



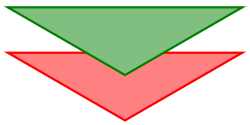
Correctness:



replay when possible

evaluate when necessary

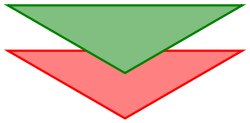
Alternating between **Evaluation** and **Replay**



a **memo** can **adapt** a **similar** execution

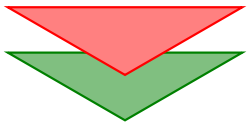
$$\frac{\overbrace{\sigma_\circ; e \Downarrow \_; \_; T_\circ}^{\textbf{previous run}} \qquad \sigma; T_\circ \curvearrowright \sigma'; v'; T'}{\sigma; \textbf{memo}\ e \Downarrow \sigma'; v'; T'}$$

Alternating between **Evaluation** and **Replay**



a **memo** can **adapt** a **similar** execution

$$\frac{\overbrace{\sigma_\circ; e \Downarrow \_; \_; T_\circ}^{\textbf{previous run}} \qquad \sigma; T_\circ \curvearrowright \sigma'; v'; T'}{\sigma; \textbf{memo}\ e \Downarrow \sigma'; v'; T'}$$



an **invalid** **unboxk** must be **reevaluated**

$$\frac{\sigma(\ell) = v \neq v_\circ \qquad \sigma; v_k\ v \Downarrow \sigma'; v'; T'}{\sigma; \underbrace{\textbf{unboxk}\ \ell\ v_\circ\ v_k}_{\textbf{record result}} \cdot T \curvearrowright \sigma'; v'; \textbf{unboxk}\ \ell\ v\ v_k \cdot T'}$$

Adaptive CPS Translation

$$[\![e^{\text{src}}]\!]\ v_k^{\text{tgt}} = e^{\text{tgt}} \quad \text{compile term } e \text{ with continuation } v_k$$

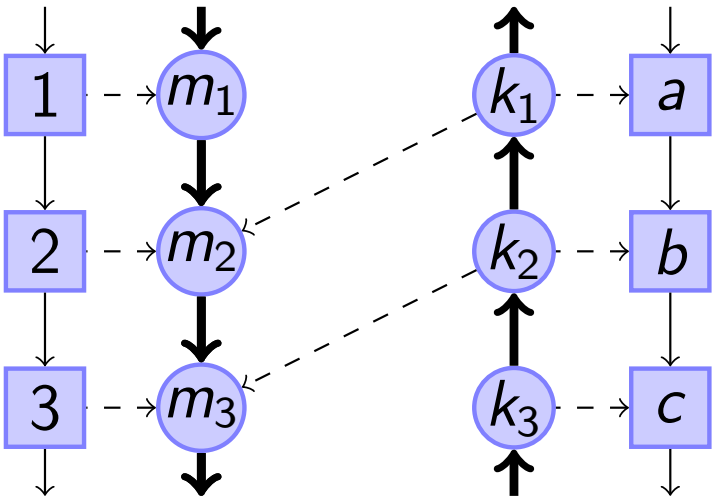
- **box** translation: straightforward

$$[\![\textbf{box}\ e]\!]\ v_k = [\![e]\!]\ (\lambda y.\textbf{boxk}\ y\ v_k)$$
$$[\![\textbf{unbox}\ e]\!]\ v_k = [\![e]\!]\ (\lambda y.\textbf{unboxk}\ y\ v_k)$$

- **fun** translation: subtle

$$[\![\textbf{fun}\ f.x.e]\!]\ v_k = \textbf{???}$$
$$[\![e_1\ e_2]\!]\ v_k = [\![e_1]\!]\ (\lambda y_f.\ [\![e_2]\!]\ (\lambda y_x.y_f\ y_x\ v_k))$$

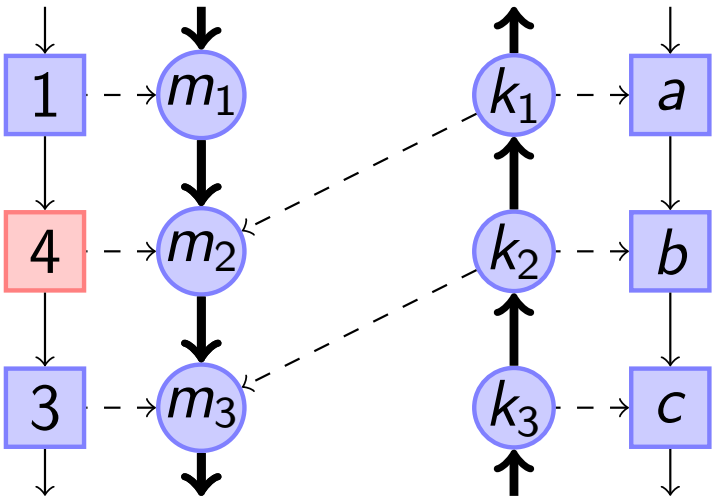
Function Compilation: standard translation



$$[\![\textbf{fun}\, f.x.e]\!]\, v_k$$
$$= v_k\ (\textbf{fun}\ f.x.k.$$

$$([\![e]\!]\ k\ ))$$

Function Compilation: standard translation

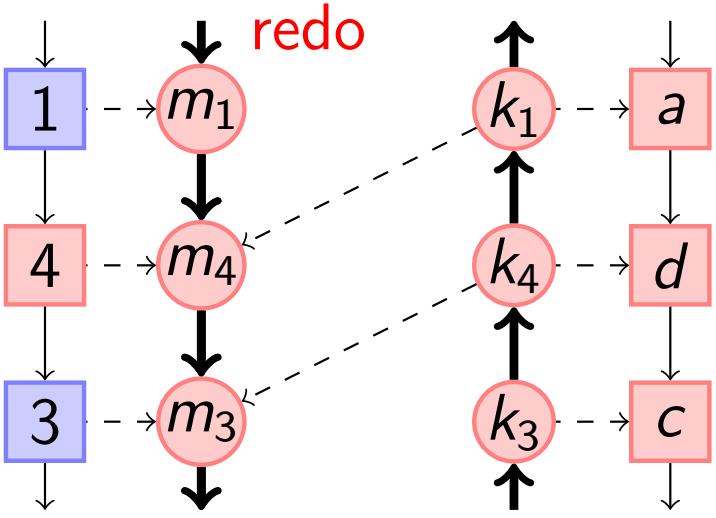


$[\![\textbf{fun}\ f.x.e]\!]\ v_k$

$= v_k\ (\textbf{fun}\ f.x.k.$

$([\![e]\!]\ k\ ))$

Function Compilation: standard translation

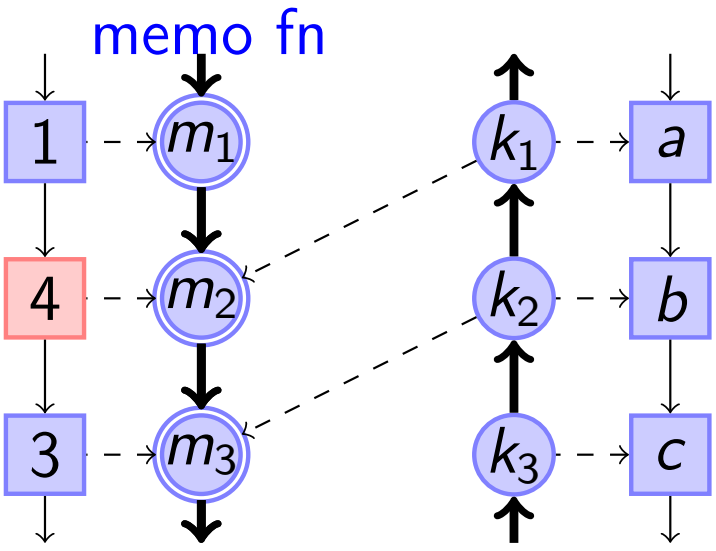


How to reuse work?

$$[\![\textbf{fun}\ f.x.e]\!]\ v_k$$
$$= v_k\ (\textbf{fun}\ f.x.k.$$

$$([\![e]\!]\ k\ ))$$

Function Compilation: memo'd call



How to reuse work?

Memo the function call

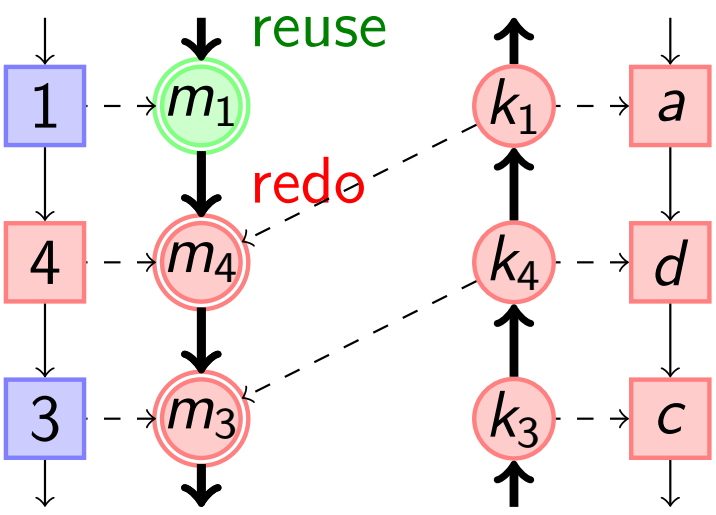
$[\![\textbf{fun}\ f.x.e]\!]\ v_k$

$= v_k\ (\textbf{fun}\ f.x.k.$

memo $([\![e]\!]\ k\ ))$

memo body

Function Compilation: memo'd call



How to reuse work?

Memo the function call

How to memo despite continuation?

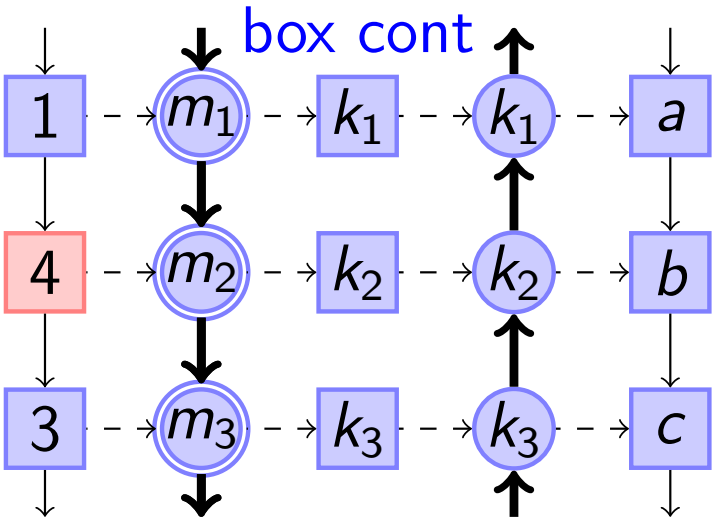
$[\![\textbf{fun}\ f.x.e]\!]\ v_k$

$= v_k\ (\textbf{fun}\ f.x.k.$

memo $([\![e]\!]\ k\ ))$

memo body

Function Compilation: memo'd call + boxed cont



How to reuse work?

Memo the function call

How to memo despite continuation?

Box the continuation

$[\![\textbf{fun}\ f.x.e]\!]\ v_k$

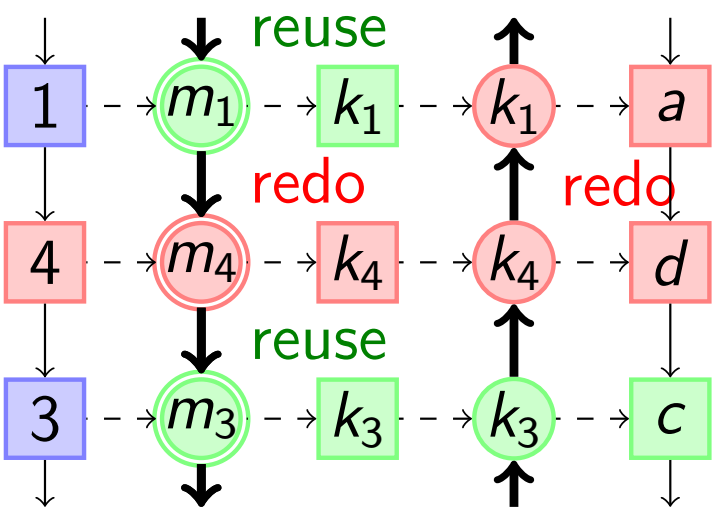
$= v_k\ (\textbf{fun}\ f.x.k.$

$\textbf{let}\ y_k = \textbf{boxk}\ k\ \textbf{in}$ **box cont**

$\textbf{let}\ k' = \lambda y_r.\textbf{unboxk}\ y_k\ (\lambda k.k\ y_r)\ \textbf{in}$ **unbox cont**

$\textbf{memo}\ ([\![e]\!]\ k'))$ **memo body**

Function Compilation: memo'd call + boxed cont



How to reuse work?

Memo the function call

How to memo despite continuation?

Box the continuation

How to avoid reexecuting continuation?

$[\![\textbf{fun}\ f.x.e]\!]\ v_k$

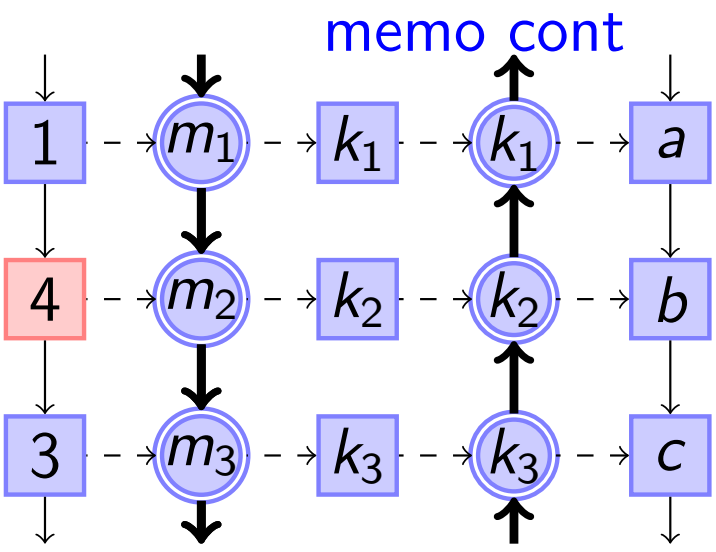
$= v_k\ (\textbf{fun}\ f.x.k.$

$\textbf{let}\ y_k = \textbf{boxk}\ k\ \textbf{in}$ **box cont**

$\textbf{let}\ k' = \lambda y_r.\textbf{unboxk}\ y_k\ (\lambda k.k\ y_r)\ \textbf{in}$ **unbox cont**

$\textbf{memo}\ ([\![e]\!]\ k'))$ **memo body**

Function Compilation: memo'd call/cont + boxed cont



How to reuse work?

Memo the function call

How to memo despite continuation?

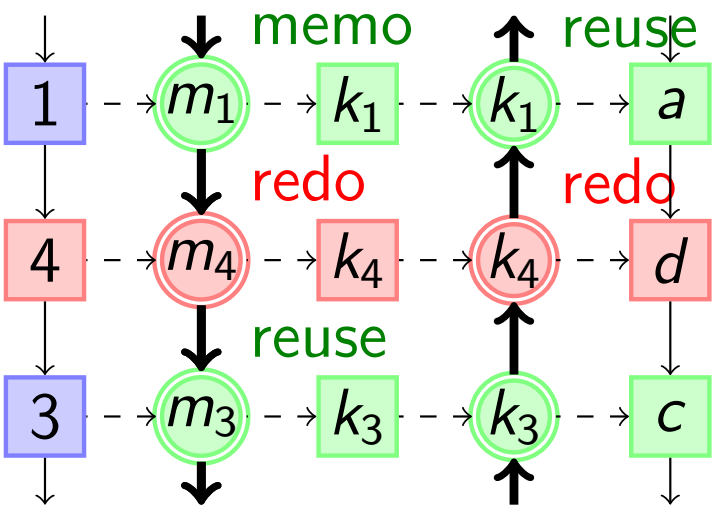
Box the continuation

How to avoid reexecuting continuation?

Memoizing continuations

$$
\begin{aligned}
&[\![\textbf{fun}\ f.x.e]\!]\ v_k \\
&\quad = v_k\ (\textbf{fun}\ f.x.k. \\
&\qquad \textbf{let}\ k_m = \lambda y_r.\textbf{memo}\ (k\ y_r)\ \textbf{in} && \textbf{memo cont} \\
&\qquad \textbf{let}\ y_k = \textbf{boxk}\ k_m\ \textbf{in} && \textbf{box cont} \\
&\qquad \textbf{let}\ k' = \lambda y_r.\textbf{unboxk}\ y_k\ (\lambda k.k\ y_r)\ \textbf{in} && \textbf{unbox cont} \\
&\qquad\quad \textbf{memo}\ ([\![e]\!]\ k')) && \textbf{memo body}
\end{aligned}
$$

Function Compilation: memo'd call/cont + boxed cont



How to reuse work?

Memo the function call

How to memo despite continuation?

Box the continuation

How to avoid reexecuting continuation?

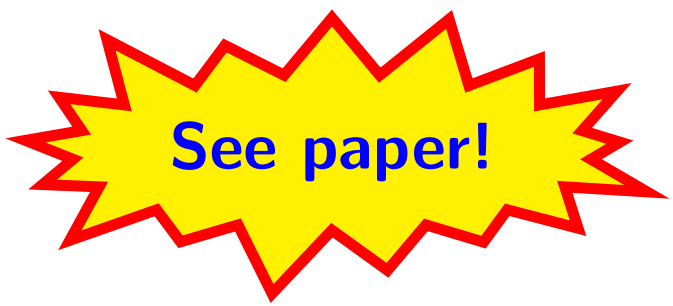
Memoizing continuations

Uses existing primitives!

$$
\begin{aligned}
&[\![\textbf{fun}\ f.x.e]\!]\ v_k \\
&\quad = v_k\ (\textbf{fun}\ f.x.k. \\
&\qquad \textbf{let}\ k_m = \lambda y_r.\textbf{memo}\ (k\ y_r)\ \textbf{in} && \textbf{memo cont} \\
&\qquad \textbf{let}\ y_k = \textbf{boxk}\ k_m\ \textbf{in} && \textbf{box cont} \\
&\qquad \textbf{let}\ k' = \lambda y_r.\textbf{unboxk}\ y_k\ (\lambda k.k\ y_r)\ \textbf{in} && \textbf{unbox cont} \\
&\qquad\quad \textbf{memo}\ ([\![e]\!]\ k')) && \textbf{memo body}
\end{aligned}
$$

Summary

- Adaptive CPS for compiling self-adjusting programs
  - Boxes = track dependencies + isolate continuations
  - Memoization = identify reuse + optimize continuations
- More natural programming style
- SML implementation in MLton
  - Selective CPS
  - Reconcile memoization with allocation
  - Interacting with self-adjusting subprograms
- Experimental evaluation
  - Efficient update
  - Competitive against manual approach



Thanks!

`http://ttic.uchicago.edu/~pl/sa-sml`

Future Work

- Reasoning principles for asymptotic performance
- Automate annotations
- Combine with effects (imperative references, I/O, ...)
- Larger applications

  Self-adjusting `latex`?