# Implicit self-adjusting computation for CostIt

## Internship Defense


Zoe Paraskevopoulou[1,2]
Advisor: Deepak Garg[2]

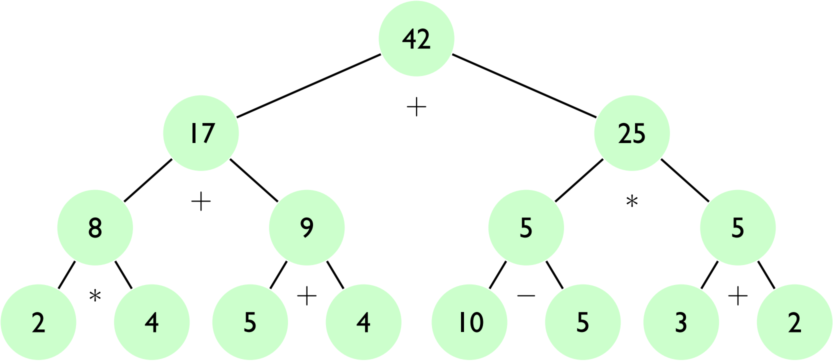
[1]ENS Cachan [2]Max Planck Institute for Software Systems


September 3, 2020

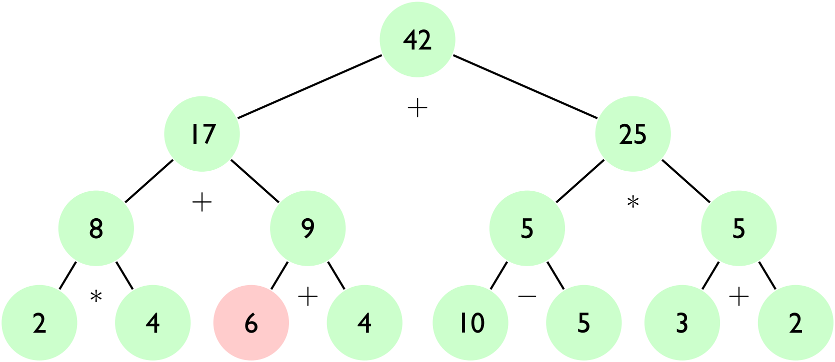
# Self Adjusting Computation

- An evaluation mechanism that recomputes *only* the parts of the output that depend on inputs that have changed between runs
- *Change propagation* (CP) : the process of updating the parts of the output that depend on changed data
- *Implicit self-adjusting computation*: The program responds automatically to changes in its inputs without any manual effort from the programmer
- Often results in asymptotic speedup

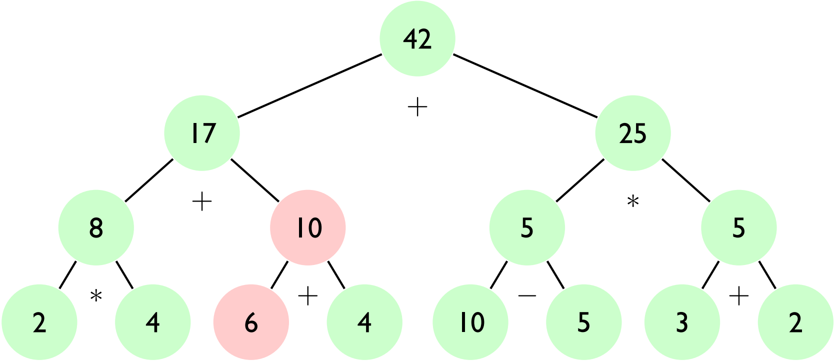
# Change Propagation by Example



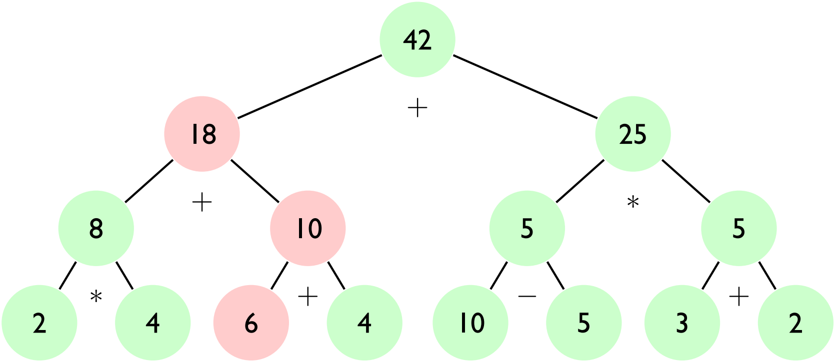
# Change Propagation by Example



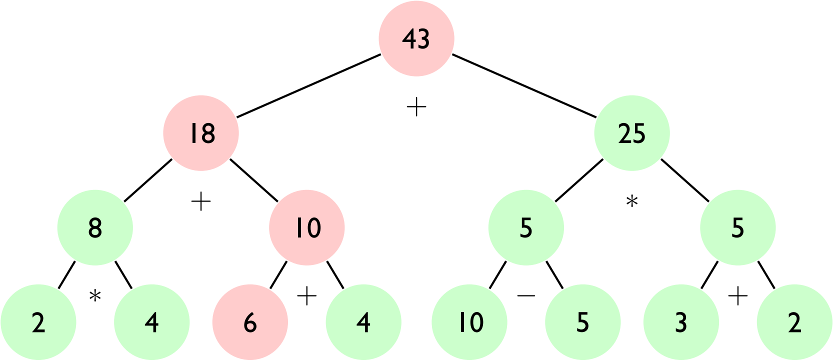
# Change Propagation by Example



# Change Propagation by Example



# Change Propagation by Example



# CostIt (I)

- A *type and effect system* that allows us to derive upper bounds on the cost of incremental computation

# CostIt (I)

- A *type and effect system* that allows us to derive upper bounds on the cost of incremental computation
- Judgments: $\Delta; \Phi; \Gamma \vdash_{\epsilon}^{\kappa} e : \tau$
  - $\epsilon$ is the typing mode ($\mathbb{S}$ or $\mathbb{C}$)
  - $\kappa$ is the derived cost

# CostIt (I)

- A *type and effect system* that allows us to derive upper bounds on the cost of incremental computation
- Judgments: $\Delta; \Phi; \Gamma \vdash_{\epsilon}^{\kappa} e : \tau$
  - $\epsilon$ is the typing mode ($\mathbb{S}$ or $\mathbb{C}$)
  - $\kappa$ is the derived cost
- When $\epsilon = \mathbb{S}$ then $\kappa$ is the upper bound of CP

# CostIt (I)

- A *type and effect system* that allows us to derive upper bounds on the cost of incremental computation
- Judgments: $\Delta; \Phi; \Gamma \vdash_{\epsilon}^{\kappa} e : \tau$
  - $\epsilon$ is the typing mode ($\mathbb{S}$ or $\mathbb{C}$)
  - $\kappa$ is the derived cost
- When $\epsilon = \mathbb{S}$ then $\kappa$ is the upper bound of CP
- When $\epsilon = \mathbb{C}$ then $\kappa$ is the worst case execution cost

# CostIt (II)

- Functions are annotated with effects $\tau_1 \xrightarrow{\mu(\kappa)} \tau_2$

# CostIt (II)

- Functions are annotated with effects $\tau_1 \xrightarrow{\mu(\kappa)} \tau_2$
  - When $\mu = \mathbb{S}$ then the result of the function application can be updated with CP with cost $\leq \kappa$

# CostIt (II)

- Functions are annotated with effects $\tau_1 \xrightarrow{\mu(\kappa)} \tau_2$
  - When $\mu = \mathbb{S}$ then the result of the function application can be updated with CP with cost $\leq \kappa$
  - When $\mu = \mathbb{C}$ then the the function application is evaluated from-scratch with cost $\leq \kappa$

# CostIt (II)

- Functions are annotated with effects $\tau_1 \xrightarrow{\mu(\kappa)} \tau_2$
  - When $\mu = \mathbb{S}$ then the result of the function application can be updated with CP with cost $\leq \kappa$
  - When $\mu = \mathbb{C}$ then the the function application is evaluated from-scratch with cost $\leq \kappa$
- Types have changeability annotations $\tau^{\mu}$
  - $\tau^{\mathbb{S}}$ : a value that cannot change between runs
  - $\tau^{\mathbb{C}}$ : a value that can change between runs
  - $\tau^{\square}$ : a value that cannot change between nor capture other changeable values

# CostIt (II)

- Functions are annotated with effects $\tau_1 \xrightarrow{\mu(\kappa)} \tau_2$
  - When $\mu = \mathbb{S}$ then the result of the function application can be updated with CP with cost $\leq \kappa$
  - When $\mu = \mathbb{C}$ then the the function application is evaluated from-scratch with cost $\leq \kappa$
- Types have changeability annotations $\tau^{\mu}$
  - $\tau^{\mathbb{S}}$ : a value that cannot change between runs
  - $\tau^{\mathbb{C}}$ : a value that can change between runs
  - $\tau^{\square}$ : a value that cannot change between nor capture other changeable values
- Index refinement types (in the style of DML)

# CostIt (II)

- Functions are annotated with effects $\tau_1 \xrightarrow{\mu(\kappa)} \tau_2$
  - When $\mu = \mathbb{S}$ then the result of the function application can be updated with CP with cost $\leq \kappa$
  - When $\mu = \mathbb{C}$ then the the function application is evaluated from-scratch with cost $\leq \kappa$
- Types have changeability annotations $\tau^{\mu}$
  - $\tau^{\mathbb{S}}$ : a value that cannot change between runs
  - $\tau^{\mathbb{C}}$ : a value that can change between runs
  - $\tau^{\square}$ : a value that cannot change between nor capture other changeable values
- Index refinement types (in the style of DML)
- Lists : $\texttt{list}\,[n]^{\alpha}\ \tau$
  - A vector of $n$ elements from which at most $\alpha$ can change

# Running Example: `map` (typing)

$$\texttt{map} : (\tau_1 \xrightarrow{\mathbb{C}(\kappa)} \tau_2)^{\square} \xrightarrow{\mathbb{S}(0)} \texttt{list}\,[n]^{\alpha}\ \tau_1 \xrightarrow{\mathbb{S}(\kappa\cdot\alpha)} \texttt{list}\,[n]^{\alpha}\ \tau_2$$

- If `f` executes from-scratch with cost $k$ and `l` has $n$ elements of which at most $\alpha$ can change then `map f l` propagates changes with cost at most $\alpha \cdot \kappa$

# Running Example: `map` (typing)

$$\texttt{map} : (\tau_1 \xrightarrow{\mathbb{C}(\kappa)} \tau_2)^{\square} \xrightarrow{\mathbb{S}(0)} \texttt{list}\,[n]^{\alpha}\ \tau_1 \xrightarrow{\mathbb{S}(\kappa\cdot\alpha)} \texttt{list}\,[n]^{\alpha}\ \tau_2$$

- If `f` executes from-scratch with cost $k$ and `l` has $n$ elements of which at most $\alpha$ can change then `map f l` propagates changes with cost at most $\alpha \cdot \kappa$
- Intuition: we need to recompute and update in place only the elements of the list that can change

# Soundness (this internship)

- **Idea:** Translate a CostIt program to a self-adjusting program and show that the actual cost is no more that the cost derived by the type system

$$\begin{tikzcd}
\vdash^{\kappa}_{\epsilon} e : \tau \arrow[rr, "\text{evaluates to}"] \arrow[d, "\text{translates to}"] & & v_1 \arrow[d, phantom, "\approx"] \\
\vdash \ulcorner e \urcorner : \|\tau\| \arrow[rr, "\text{evaluates in } c \text{ steps to}", "\text{if } \epsilon = \mathbb{C} \text{ then } c \leq \kappa"'] & & v_2
\end{tikzcd}$$

(a) First run

$$\begin{tikzcd}
\vdash^{\kappa}_{\mathbb{S}} e : \tau \arrow[rr, "\text{evaluates to}"] \arrow[d, "\text{translates to}"] & & v_1 \arrow[d, phantom, "\approx"] \\
\vdash \ulcorner e \urcorner : \|\tau\| \arrow[rr, "\text{CP in } c \text{ steps to}", "c \leq \kappa"'] & & v_2
\end{tikzcd}$$

(b) Incremental run after input changes

Figure: Schematic representation of the basic properties of the translation

# Target Language: saML

- A simply typed lambda calculus with general references

# Target Language: saML

- A simply typed lambda calculus with general references
- The runtime is modified to keep track of the computations that need to be re-executed during CP

# Target Language: saML

- A simply typed lambda calculus with general references
- The runtime is modified to keep track of the computations that need to be re-executed during CP
    - We maintain a global queue that holds closures that are pushed during the first run

# Target Language: saML

- A simply typed lambda calculus with general references
- The runtime is modified to keep track of the computations that need to be re-executed during CP
  - We maintain a global queue that holds closures that are pushed during the first run
  - We add new primitives: `push`, `empty`

# Target Language: saML

- A simply typed lambda calculus with general references
- The runtime is modified to keep track of the computations that need to be re-executed during CP
  - We maintain a global queue that holds closures that are pushed during the first run
  - We add new primitives: `push`, `empty`
  - We push tuples of the form $(\vec{l},\ f)$, where $\vec{l}$ is the list of locations that need to be updated and $f$ the closure that computes their new values

# Target Language: saML

- A simply typed lambda calculus with general references
- The runtime is modified to keep track of the computations that need to be re-executed during CP
    - We maintain a global queue that holds closures that are pushed during the first run
    - We add new primitives: `push`, `empty`
    - We push tuples of the form $(\vec{l},\ f)$, where $\vec{l}$ is the list of locations that need to be updated and $f$ the closure that computes their new values
    - During the incremental run the computations are popped and executed with a FIFO order

# Target Language: saML

- A simply typed lambda calculus with general references
- The runtime is modified to keep track of the computations that need to be re-executed during CP
  - We maintain a global queue that holds closures that are pushed during the first run
  - We add new primitives: `push`, `empty`
  - We push tuples of the form $(\vec{l},\ f)$, where $\vec{l}$ is the list of locations that need to be updated and $f$ the closure that computes their new values
  - During the incremental run the computations are popped and executed with a FIFO order

# Running Example: `map` (translation I)

$$\texttt{map} : (\texttt{int}^{\mathbb{C}} \xrightarrow{\mathbb{C}(\kappa)} \texttt{int}^{\mathbb{C}})^{\square} \xrightarrow{\mathbb{S}(0)} \texttt{list}\,[n]^{\alpha}\ \texttt{int}^{\mathbb{C}} \xrightarrow{\mathbb{S}(\kappa\cdot\alpha)} \texttt{list}\,[n]^{\alpha}\ \texttt{int}^{\mathbb{C}}$$

- Re-apply the argument function only to the elements that have changed and update the output list in-place

# Running Example: `map` (translation I)

$$\texttt{map} : (\texttt{int}^{\mathbb{C}} \xrightarrow{\mathbb{C}(\kappa)} \texttt{int}^{\mathbb{C}})^{\Box} \xrightarrow{\mathbb{S}(0)} \texttt{list}\,[n]^{\alpha}\ \texttt{int}^{\mathbb{C}} \xrightarrow{\mathbb{S}(\kappa\cdot\alpha)} \texttt{list}\,[n]^{\alpha}\ \texttt{int}^{\mathbb{C}}$$

- Re-apply the argument function only to the elements that have changed and update the output list in-place
- Store changeable values in reference cells : $\|A^{\mathbb{C}}\| = \texttt{ref}\ \|A\|$

# Running Example: `map` (translation I)

$$\texttt{map} : (\texttt{int}^{\mathbb{C}} \xrightarrow{\mathbb{C}(\kappa)} \texttt{int}^{\mathbb{C}})^{\square} \xrightarrow{\mathbb{S}(0)} \texttt{list}\,[n]^{\alpha}\ \texttt{int}^{\mathbb{C}} \xrightarrow{\mathbb{S}(\kappa\cdot\alpha)} \texttt{list}\,[n]^{\alpha}\ \texttt{int}^{\mathbb{C}}$$

- Re-apply the argument function only to the elements that have changed and update the output list in-place
- Store changeable values in reference cells : $\|A^{\mathbb{C}}\| = \texttt{ref}\ \|A\|$
- Differentiate between stable and changeable values of a list : $\|\texttt{list}\,[n]^{\alpha}\ \texttt{int}^{\mathbb{C}}\| = \texttt{list}\ (\texttt{int} + \texttt{ref int})$

# Running Example: map (translation I)

$$\texttt{map} : (\texttt{int}^{\mathbb{C}} \xrightarrow{\mathbb{C}(\kappa)} \texttt{int}^{\mathbb{C}})^{\square} \xrightarrow{\mathbb{S}(0)} \texttt{list}\,[n]^{\alpha}\ \texttt{int}^{\mathbb{C}} \xrightarrow{\mathbb{S}(\kappa\cdot\alpha)} \texttt{list}\,[n]^{\alpha}\ \texttt{int}^{\mathbb{C}}$$

- Re-apply the argument function only to the elements that have changed and update the output list in-place
- Store changeable values in reference cells : $\|A^{\mathbb{C}}\| = \texttt{ref}\ \|A\|$
- Differentiate between stable and changeable values of a list : $\|\texttt{list}\,[n]^{\alpha}\ \texttt{int}^{\mathbb{C}}\| = \texttt{list}\ (\texttt{int} + \texttt{ref int})$

$$\ulcorner\texttt{map}\urcorner : (\texttt{ref int} \rightarrow \texttt{ref int}) \rightarrow \texttt{list}\ (\texttt{int} + \texttt{ref int}) \rightarrow \texttt{list}\ (\texttt{int} + \texttt{ref int})$$

# Running Example: `map` (translation II)

$$\ulcorner \texttt{map} \urcorner : (\texttt{ref int} \to \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int})$$

$$\begin{array}{l} \texttt{map}\ f\ e = \\ \quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of} \\ \quad \mid [\,] \ \to\ [\,] \\ \quad \mid h :: tl \ \to\ f\ h :: \texttt{map}\ f\ tl \end{array}$$

$$\begin{array}{l} \ulcorner \texttt{map} \urcorner\ f\ e = \\ \quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of} \\ \quad \mid [\,] \ \to \\ \quad \mid h :: tl \ \to \end{array}$$

# Running Example: `map` (translation II)

$$\ulcorner \texttt{map} \urcorner : (\texttt{ref int} \to \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int})$$

$$\begin{array}{l} \texttt{map}\ f\ e = \\ \quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of} \\ \quad \mid [\,] \ \to\ [\,] \\ \quad \mid h :: tl \ \to\ f\ h :: \texttt{map}\ f\ tl \end{array}$$

$$\begin{array}{l} \ulcorner \texttt{map} \urcorner\ f\ e = \\ \quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of} \\ \quad \mid [\,] \ \to\ [\,] \\ \quad \mid h :: tl \ \to \end{array}$$

# Running Example: `map` (translation II)

$$\ulcorner \texttt{map} \urcorner : (\texttt{ref int} \to \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int})$$

$$
\begin{array}{l}
\texttt{map}\ f\ e = \\
\quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of} \\
\quad \mid [\,] \ \to\ [\,] \\
\quad \mid h :: tl\ \to\ f\ h :: \texttt{map}\ f\ tl
\end{array}
$$

$$
\begin{array}{l}
\ulcorner \texttt{map} \urcorner\ f\ e = \\
\quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of} \\
\quad \mid [\,] \ \to\ [\,] \\
\quad \mid h :: tl\ \to \\
\qquad \texttt{case}\ h\ \texttt{of} \\
\qquad \mid h_l\ \to \\
\qquad \mid h_r\ \to
\end{array}
$$

# Running Example: `map` (translation II)

$$\ulcorner\texttt{map}\urcorner : (\texttt{ref int} \to \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int})$$

$$
\begin{array}{l}
\texttt{map}\ f\ e =\\
\quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of}\\
\quad \mid [\,]\ \to\ [\,]\\
\quad \mid h :: tl\ \to f\ h :: \texttt{map}\ f\ tl
\end{array}
$$

$$
\begin{array}{l}
\ulcorner\texttt{map}\urcorner\ f\ e =\\
\quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of}\\
\quad \mid [\,]\ \to\ [\,]\\
\quad \mid h :: tl\ \to\\
\qquad \texttt{case}\ h\ \texttt{of}\\
\qquad \mid h_l\ \to\ (\texttt{inl}\ !(f\ (\texttt{ref}\ h_l))) :: \ulcorner\texttt{map}\urcorner\ f\ tl\\
\qquad \mid h_r\ \to
\end{array}
$$

# Running Example: `map` (translation II)

$$\ulcorner\texttt{map}\urcorner : (\texttt{ref int} \to \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int})$$

$$\begin{array}{l} \texttt{map}\ f\ e = \\ \quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of} \\ \quad \mid [\,] \ \to\ [\,] \\ \quad \mid h :: tl \ \to f\ h :: \texttt{map}\ f\ tl \end{array}$$

$$\begin{array}{l} \ulcorner\texttt{map}\urcorner\ f\ e = \\ \quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of} \\ \quad \mid [\,] \ \to\ [\,] \\ \quad \mid h :: tl \ \to \\ \qquad \texttt{case}\ h\ \texttt{of} \\ \qquad \mid h_l \ \to\ (\texttt{inl}\ !(f\ (\texttt{ref}\ h_l))) :: \ulcorner\texttt{map}\urcorner\ f\ tl \\ \qquad \mid h_r \ \to\ \texttt{let}\ l\ =\ \texttt{ref}\ !(f\ h_r)\ \texttt{in} \end{array}$$

# Running Example: `map` (translation II)

$$\ulcorner \texttt{map} \urcorner : (\texttt{ref int} \to \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int})$$

$$
\begin{array}{l}
\texttt{map}\ f\ e = \\
\quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of} \\
\quad \mid [] \ \to\ [] \\
\quad \mid h :: tl \ \to f\ h :: \texttt{map}\ f\ tl
\end{array}
$$

$$
\begin{array}{l}
\ulcorner \texttt{map} \urcorner\ f\ e = \\
\quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of} \\
\quad \mid [] \ \to\ [] \\
\quad \mid h :: tl \ \to \\
\qquad \texttt{case}\ h\ \texttt{of} \\
\qquad \mid h_l \ \to\ (\texttt{inl}\ !(f\ (\texttt{ref}\ h_l))) :: \ulcorner \texttt{map} \urcorner\ f\ tl \\
\qquad \mid h_r \ \to\ \texttt{let}\ l\ =\ \texttt{ref}\ !(f\ h_r)\ \texttt{in} \\
\qquad \qquad\quad\ \texttt{let}\ ()\ = \\
\qquad \qquad\qquad \texttt{push}(l,\ \lambda().!(f\ h_r))\ \texttt{in}
\end{array}
$$

# Running Example: `map` (translation II)

$$\ulcorner \texttt{map} \urcorner : (\texttt{ref int} \to \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int}) \to \texttt{list}\,(\texttt{int} + \texttt{ref int})$$

$$
\begin{array}{l}
\texttt{map}\ f\ e = \\
\quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of} \\
\quad \mid [] \ \to\ [] \\
\quad \mid h :: tl\ \to f\ h :: \texttt{map}\ f\ tl
\end{array}
$$

$$
\begin{array}{l}
\ulcorner \texttt{map} \urcorner\ f\ e = \\
\quad \texttt{case}_{\mathrm{L}}\ e\ \texttt{of} \\
\quad \mid [] \ \to\ [] \\
\quad \mid h :: tl\ \to \\
\qquad \texttt{case}\ h\ \texttt{of} \\
\qquad \mid h_l\ \to\ (\texttt{inl}\ !(f\ (\texttt{ref}\ h_l))) :: \ulcorner \texttt{map} \urcorner\ f\ tl \\
\qquad \mid h_r\ \to\ \texttt{let}\ l\ =\ \texttt{ref}\ !(f\ h_r)\ \texttt{in} \\
\qquad \qquad\quad \texttt{let}\ ()\ = \\
\qquad \qquad\qquad \texttt{push}(l,\ \lambda().!(f\ h_r))\ \texttt{in} \\
\qquad \qquad\quad \texttt{inr}\ l :: \ulcorner \texttt{map} \urcorner\ f\ tl
\end{array}
$$

# Translation

$$\Delta; \Phi; \Gamma \vdash_{\epsilon}^{\kappa} e : \tau \hookrightarrow \ulcorner e \urcorner$$

# Translation

$$\Delta; \Phi; \Gamma \vdash_{\epsilon}^{\kappa} e : \tau \hookrightarrow \ulcorner e \urcorner$$

- The translation is defined by induction on the typing derivation

# Translation

$$\Delta; \Phi; \Gamma \vdash_{\epsilon}^{\kappa} e : \tau \hookrightarrow \ulcorner e \urcorner$$

- The translation is defined by induction on the typing derivation
- Two modes: $\epsilon = \mathbb{C}$ and $\epsilon = \mathbb{S}$

# Translation

$$\Delta; \Phi; \Gamma \vdash_{\epsilon}^{\kappa} e : \tau \hookrightarrow \ulcorner e \urcorner$$

- The translation is defined by induction on the typing derivation
- Two modes: $\epsilon = \mathbb{C}$ and $\epsilon = \mathbb{S}$
- The code generated in $\mathbb{C}$ mode will be executed from scratch during CP

# Translation

$$\Delta; \Phi; \Gamma \vdash_{\epsilon}^{\kappa} e : \tau \hookrightarrow \ulcorner e \urcorner$$

- The translation is defined by induction on the typing derivation
- Two modes: $\epsilon = \mathbb{C}$ and $\epsilon = \mathbb{S}$
- The code generated in $\mathbb{C}$ mode will be executed from scratch during CP
- The code generated in $\mathbb{S}$ mode is self-adjusting
  - During this mode we record the computations that need to be re-executed during CP

# Change Propagation

$$Q,\ \sigma_i[\sigma_c] \rightsquigarrow \sigma_f,\ c$$

# Change Propagation

$$Q,\ \sigma_i[\sigma_c] \rightsquigarrow \sigma_f,\ c$$

- $Q$ is the queue holding the recorded computations

# Change Propagation

$$Q,\ \sigma_i[\sigma_c] \rightsquigarrow \sigma_f,\ c$$

- $Q$ is the queue holding the recorded computations
- While $Q$ is not empty, the algorithm:

# Change Propagation

$$Q,\ \sigma_i[\sigma_c] \leadsto \sigma_f,\ c$$

- $Q$ is the queue holding the recorded computations
- While $Q$ is not empty, the algorithm:
  - pops an element $(\vec{l},\ f)$ from the queue

# Change Propagation

$$Q,\ \sigma_i[\sigma_c] \rightsquigarrow \sigma_f,\ c$$

- $Q$ is the queue holding the recorded computations
- While $Q$ is not empty, the algorithm:
    - pops an element $(\vec{l},\ f)$ from the queue
    - runs the computation $f\ ()$ that returns the updated values of the locations and incurs cost $c_f$
    - updates the locations with their new values and the total cost to $c \leftarrow c_f + c$

# Change Propagation

$$Q,\ \sigma_i[\sigma_c] \rightsquigarrow \sigma_f,\ c$$

- $Q$ is the queue holding the recorded computations
- While $Q$ is not empty, the algorithm:
  - pops an element $(\vec{l},\ f)$ from the queue
  - runs the computation $f\ ()$ that returns the updated values of the locations and incurs cost $c_f$
  - updates the locations with their new values and the total cost to $c \leftarrow c_f + c$

# Similarity Relation

$$v_s \approx^{\tau}_{\sigma} v_t$$

- $v_s$ is the source value, $v_t$ is the target value
- $\sigma$ is the store in the target
- Changeable values are references in the target (stored in $\sigma$)
- For *stable values*, $v_s$ and $v_t$ should coincide
- For *changeable values*, $v_t$ should be a location and $v_s$ should coincide with the value of this location in the store.

$$(3, 42) \approx^{\mathtt{int}^{\mathbb{S}} \times \mathtt{int}^{\mathbb{C}}}_{[l \mapsto 42]} (3, l)$$

# Soundness, $\mathbb{C}$ mode

## Theorem

*Assume that*

$$\vdash^{\kappa}_{\mathbb{C}} e : \tau \hookrightarrow \ulcorner e \urcorner$$

*Then there exist $v'_s$, $v'_t$, $\sigma'$, $j$ and $c$, such that*

# Soundness, $\mathbb{C}$ mode

## Theorem

*Assume that*

$$\vdash^{\kappa}_{\mathbb{C}} e : \tau \hookrightarrow \ulcorner e \urcorner$$

*Then there exist* $v'_s$, $v'_t$, $\sigma'$, $j$ *and* $c$, *such that*

(1) $e \Downarrow v'_s,\ j$

# Soundness, $\mathbb{C}$ mode

**Theorem**

*Assume that*

$$\vdash^{\kappa}_{\mathbb{C}} e : \tau \hookrightarrow \ulcorner e \urcorner$$

*Then there exist $v'_s$, $v'_t$, $\sigma'$, $j$ and $c$, such that*

(1) $e \Downarrow v'_s,\ j$

(2) $\ulcorner e \urcorner,\ \sigma \Downarrow v'_t,\ \sigma',\ \varnothing,\ c$

# Soundness, $\mathbb{C}$ mode

**Theorem**

*Assume that*

$$\vdash^{\kappa}_{\mathbb{C}} e : \tau \hookrightarrow \ulcorner e \urcorner$$

*Then there exist* $v'_s$, $v'_t$, $\sigma'$, $j$ *and* $c$*, such that*

(1) $e \Downarrow v'_s,\ j$

(2) $\ulcorner e \urcorner,\ \sigma \Downarrow v'_t,\ \sigma',\ \varnothing,\ c$

(3) $\models c \mathrel{\dot{\leq}} \kappa$

# Soundness, $\mathbb{C}$ mode

**Theorem**

*Assume that*

$$\vdash^{\kappa}_{\mathbb{C}} e : \tau \hookrightarrow \ulcorner e \urcorner$$

*Then there exist* $v'_s$, $v'_t$, $\sigma'$, $j$ *and* $c$*, such that*

(1) $e \Downarrow v'_s,\ j$

(2) $\ulcorner e \urcorner,\ \sigma \Downarrow v'_t,\ \sigma',\ \varnothing,\ c$

(3) $\models c \,\dot{\leq}\, \kappa$

(4) $v'_s \approx^{\tau}_{\sigma'} v'_t$

# Two-way similarity relation

$$(v_i,\ v_c) \approx^{\tau}_{(\sigma_i,\ \sigma_c)} v_t$$

- $v_i$ is the initial source value, $v_c$ is the source value after changes
- $v_t$ is the target value that stores changeable values in references
- $\sigma_i$ is the initial target store, $\sigma_c$ is the target store holding changed values
- For *stable values*, $v_i$, $v_c$ and $v_t$ should coincide under the two stores
- For *changeable values*, $v_i$ should be similar to $v_t$ under $\sigma_i$ and $v_c$ should be similar to $v_t$ under $\sigma_c$

$$((3, 42),\ (3, 43)) \approx^{\mathtt{int}^{\mathbb{S}} \times \mathtt{int}^{\mathbb{C}}}_{[l \mapsto 42],\ [l \mapsto 43]} (3, l)$$

# Soundness, $\mathbb{S}$ mode

## Theorem

*Assume that*

$$\cdot;\cdot;x:\tau' \vdash^{\kappa}_{\mathbb{S}} e:\tau \hookrightarrow \ulcorner e \urcorner$$

$$(v_i,\ v_c) \approx^{\tau'}_{(\sigma_i,\ \sigma_c)} v_t$$

*Then if* $[x \mapsto v_i]e \Downarrow v_i',\ j$ *then there exist* $v_c'$, $v_t'$, $\sigma_f$, $\sigma_f'$, $Q$, $j$ *and* $c$, *such that*

# Soundness, $\mathbb{S}$ mode

## Theorem

*Assume that*

$$\cdot;\cdot;x:\tau' \vdash^{\kappa}_{\mathbb{S}} e:\tau \hookrightarrow \ulcorner e \urcorner$$

$$(v_i,\ v_c) \approx^{\tau'}_{(\sigma_i,\ \sigma_c)} v_t$$

*Then if* $[x \mapsto v_i]e \Downarrow v'_i,\ j$ *then there exist* $v'_c$, $v'_t$, $\sigma_f$, $\sigma'_f$, $Q$, $j$ *and* $c$, *such that*

(1) $[x \mapsto v_c]e \Downarrow v'_c,\ j'$

# Soundness, $\mathbb{S}$ mode

## Theorem

*Assume that*

$$\cdot; \cdot; x : \tau' \vdash^{\kappa}_{\mathbb{S}} e : \tau \hookrightarrow \ulcorner e \urcorner$$

$$(v_i,\ v_c) \approx^{\tau'}_{(\sigma_i,\ \sigma_c)} v_t$$

*Then if* $[x \mapsto v_i]e \Downarrow v_i',\ j$ *then there exist* $v_c'$*,* $v_t'$*,* $\sigma_f$*,* $\sigma_f'$*,* $Q$*,* $j$ *and* $c$*, such that*

(1) $[x \mapsto v_c]e \Downarrow v_c',\ j'$

(2) $[x \mapsto v_t]\ulcorner e \urcorner,\ \sigma_i \Downarrow v_t',\ \sigma_f,\ Q,\ c$

# Soundness, $\mathbb{S}$ mode

## Theorem

*Assume that*

$$\cdot;\cdot;x:\tau' \vdash^{\kappa}_{\mathbb{S}} e:\tau \hookrightarrow \ulcorner e \urcorner$$

$$(v_i,\ v_c) \approx^{\tau'}_{(\sigma_i,\ \sigma_c)} v_t$$

*Then if* $[x \mapsto v_i]e \Downarrow v_i',\ j$ *then there exist* $v_c'$, $v_t'$, $\sigma_f$, $\sigma_f'$, $Q$, $j$ *and* $c$, *such that*

(1) $[x \mapsto v_c]e \Downarrow v_c',\ j'$

(2) $[x \mapsto v_t]\ulcorner e \urcorner,\ \sigma_i \Downarrow v_t',\ \sigma_f,\ Q,\ c$

(3) $Q,\ \sigma_f[\sigma_c] \rightsquigarrow \sigma_f',\ c'$

# Soundness, $\mathbb{S}$ mode

## Theorem

*Assume that*

$$\cdot; \cdot; x : \tau' \vdash^{\kappa}_{\mathbb{S}} e : \tau \hookrightarrow \ulcorner e \urcorner$$

$$(v_i,\ v_c) \approx^{\tau'}_{(\sigma_i,\ \sigma_c)} v_t$$

*Then if* $[x \mapsto v_i]e \Downarrow v_i',\ j$ *then there exist* $v_c'$, $v_t'$, $\sigma_f$, $\sigma_f'$, $Q$, $j$ *and* $c$, *such that*

(1) $[x \mapsto v_c]e \Downarrow v_c',\ j'$

(2) $[x \mapsto v_t]\ulcorner e \urcorner,\ \sigma_i \Downarrow v_t',\ \sigma_f,\ Q,\ c$

(3) $Q,\ \sigma_f[\sigma_c] \rightsquigarrow \sigma_f',\ c'$

(4) $\models c' \mathrel{\dot{\leq}} \kappa$

# Soundness, $\mathbb{S}$ mode

## Theorem

*Assume that*

$$\cdot; \cdot; x : \tau' \vdash^{\kappa}_{\mathbb{S}} e : \tau \hookrightarrow \ulcorner e \urcorner$$

$$(v_i,\ v_c) \approx^{\tau'}_{(\sigma_i,\ \sigma_c)} v_t$$

*Then if* $[x \mapsto v_i]e \Downarrow v_i',\ j$ *then there exist* $v_c'$, $v_t'$, $\sigma_f$, $\sigma_f'$, $Q$, $j$ *and* $c$, *such that*

(1) $[x \mapsto v_c]e \Downarrow v_c',\ j'$

(2) $[x \mapsto v_t]\ulcorner e \urcorner,\ \sigma_i \Downarrow v_t',\ \sigma_f,\ Q,\ c$

(3) $Q,\ \sigma_f[\sigma_c] \rightsquigarrow \sigma_f',\ c'$

(4) $\models c' \,\dot{\leq}\, \kappa$

(5) $(v_i',\ v_c') \approx_{(\sigma_f,\ \sigma_f')} v_t'$

# Proof Method

- The soundness is proved using logical relations
- We construct two Kripke step-indexed relational models
- Two fundamental properties, one for each typing mode
- The soundness theorems are corollaries of the fundamental properties of the logical relations

# Summary

- Soundness proof for CostIt w.r.t. to concrete CP semantics
  - Older poof was w.r.t. an abstract semantics
- Designed a target language (saML) with infrastructure for CP
- Translated CostIt to saML
- Proved the correctness of the translation and the change propagation mechanism
- Proved that the cost derived by CostIt is a sound approximation of the actual cost (for both $\mathbb{C}$ and $\mathbb{S}$ modes)

# Future Work

- Devise a more efficient CP mechanism
- Mechanize the proof using a proof assistant
- Adapt CostIt to derive the cost for demand-driven self-adjusting computation
- Ongoing work: Implementation of the type system using bidirectional type checking (E. Çiçek and D. Garg)

# Thank You!

Questions?