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 8, 2015
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

![Diagram](Image)

- The diagram represents a tree structure where each node is connected by paths of addition (+) and multiplication (∗).
- The root node is labeled 42, and it has two child nodes, one labeled 17 and the other labeled 25.
- The node 17 has child nodes labeled 8, 4, 5, and 9, while the node 25 has child nodes labeled 5, 3, and 5.
- The final result, 42, is achieved by adding the values from the child nodes of the root node.

Mathematically, the expression can be represented as:

\[ 42 = (17 + (8 \times 5 + 9)) + (25 \times (5 - 3 + 2)) \]
Change Propagation by Example

\[
\begin{align*}
42 &= 17 + 25 \\
17 &= 8 + 9 \\
8 &= 2 \times 4 \\
9 &= 6 + 4 \\
25 &= 5 \times 5 \\
5 &= 10 - 5 \\
10 &= 3 + 2
\end{align*}
\]
Change Propagation by Example

```
42
/ \                      /
17  25                   +
/ \ +                    +
8  10  5                 +
/ \  / \                  +
2  4  6  4               +
/ \  / \  / \              +
2* 4* 6* 4               +
```

3 / 20
Change Propagation by Example

\[
\begin{align*}
42 &= 18 + 25 \\
18 &= 8 + 10 \\
8 &= 2 \times 4 \\
10 &= 6 + 4 \\
5 &= 5 - 5 \\
5 &= 3 + 2
\end{align*}
\]
Change Propagation by Example

\[ \frac{3}{20} \]

\[ 43 = 18 + 25 \]

\[ 18 = 8 + 10 \]

\[ 8 = 2 \times 4 \]

\[ 10 = 6 + 4 \]

\[ 6 = 2 \times 3 \]

\[ 4 = 10 - 5 \]

\[ 5 = 3 + 2 \]
• A *type and effect system* that allows us to derive upper bounds on the cost of incremental computation
Costlt (l)

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

• When $\epsilon = S$ 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:** $\Delta; \Phi; \Gamma \vdash^\kappa_\epsilon e : \tau$
  - $\epsilon$ is the typing mode ($\Sigma$ or $\mathbb{C}$).
  - $\kappa$ is the derived cost.
- When $\epsilon = \Sigma$ 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 $\tau_1 \xrightarrow{\mu(\kappa)} \tau_2$

- Types have changeability annotations
  - $\tau_S$: a value that cannot change between runs
  - $\tau_C$: 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: $\text{list}[n]^{\alpha} \tau$: A vector of $n$ elements from which at most $\alpha$ can change.
CostIt (II)

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

- Types have changeability annotations
  - $\tau_S$: a value that cannot change between runs
  - $\tau_C$: a value that can change between runs
  - $\tau_{\Box}$: a value that cannot change between nor capture other changeable values

- Index refinement types (in the style of DML)
  - Lists: $\text{list}[n]_{\alpha}$: a vector of $n$ elements from which at most $\alpha$ can change
CostIt (II)

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

- Types have changeability annotations $\tau^\mu$
  - ♦ $\tau^S$: a value that cannot change between runs
  - ♦ $\tau^C$: a value that can change between runs
  - ♦ $\tau^{\Box}$: a value that cannot change between nor capture other changeable values

Index refinement types (in the style of DML)

- Lists: $\text{list}\{n\}_{\alpha}\tau$
  - A vector of $n$ elements from which at most $\alpha$ can change
CostIt (II)

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

- Types have changeability annotations $\tau^\mu$
  - $\tau^S$: a value that cannot change between runs
  - $\tau^C$: 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)
CostLit (II)

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

- Types have changeability annotations $\tau^\mu$
  - $\tau^S$: a value that cannot change between runs
  - $\tau^C$: a value that can change between runs
  - $\tau^\Box$: a value that cannot change between nor capture other changeable values

- Index refinement types (in the style of DML)
- Lists: $\text{list}[n]^\alpha \tau$
  - A vector of $n$ elements from which at most $\alpha$ can change
Running Example: map (typing)

\[
\text{map} : (\tau_1 \xrightarrow{C(\kappa)} \tau_2) \xrightarrow{\square} \text{list}[n]^\alpha \xrightarrow{S(\kappa \cdot \alpha)} \text{list}[n]^\alpha \tau_2
\]

- If \( f \) executes from-scratch with cost \( k \) and \( l \) has \( n \) elements of which at most \( \alpha \) can change then \( \text{map} f \ l \) propagates changes with cost at most \( \alpha \cdot \kappa \).
Running Example: map (typing)

\[
\text{map} : (\tau_1 \xrightarrow{C(\kappa)} \tau_2) \xrightarrow{S(0)} \text{list}[n]^{\alpha} \tau_1 \xrightarrow{S(\kappa \cdot \alpha)} \text{list}[n]^{\alpha} \tau_2
\]

- 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

\[ \vdash_{\kappa} e : \tau \quad \text{evaluates to} \quad v_1 \]

\[ \vdash_{\kappa} e : \tau \quad \text{evaluates to} \quad v_1 \]

\[ \vdash_{\kappa} e : \|\tau\| \quad \text{evaluates in } c \text{ steps to} \quad v_2 \]

\[ \vdash_{\kappa} e : \|\tau\| \quad \text{evaluates in } c \text{ steps to} \quad v_2 \]

\[ \vdash_{\kappa} e : \|\tau\| \quad \text{evaluates in } c \text{ steps to} \quad v_2 \]

\[ \vdash_{\kappa} e : \|\tau\| \quad \text{evaluates in } c \text{ steps to} \quad v_2 \]

\[ \vdash_{\kappa} e : \|\tau\| \quad \text{evaluates in } c \text{ steps to} \quad v_2 \]

\[ \vdash_{\kappa} e : \|\tau\| \quad \text{evaluates in } c \text{ steps to} \quad v_2 \]

\[ \vdash_{\kappa} e : \|\tau\| \quad \text{evaluates in } c \text{ steps to} \quad v_2 \]

Figure: Schematic representation of the basic properties of the translation.
• A simply typed lambda calculus with general references
• 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
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
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
• 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
• 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
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: $\text{map}$ (translation I)

\[
\text{map} : (\text{int}_C \xrightarrow{\text{C}(\kappa)} \text{int}_C) \xrightarrow{\text{S}(0)} \text{list}[n]^\alpha \text{int}_C \xrightarrow{\text{S}(\kappa \cdot \alpha)} \text{list}[n]^\alpha \text{int}_C
\]

- Re-apply the argument function only to the elements that have changed and update the output list in-place
Running Example: \( \text{map} \) (translation I)

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

- 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: \( \| A^{\mathbb{C}} \| = \text{ref} \| A \| \)
Running Example: \texttt{map} (translation I)

\texttt{map} : \left( \text{int}^\text{C} \xrightarrow{\text{C(}\kappa\text{)}} \text{int}^\text{C} \right) \square \xrightarrow{S(0)} \text{list}[^n\alpha\text{int}^\text{C}} \xrightarrow{S(\kappa\cdot\alpha)} \text{list}[^n\alpha\text{int}^\text{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: \( \| A^\text{C} \| = \text{ref} \| A \| \)
- Differentiate between stable and changeable values of a list:
  \( \| \text{list}[^n\alpha\text{int}^\text{C}} \| = \text{list (int + ref int)} \)
Running Example: map (translation I)

\[ \text{map} : (\text{int}^C \xrightarrow{\kappa} \text{int}^C) \xrightarrow{\Box} \text{list}^\alpha \text{int}^C \xrightarrow{\kappa \cdot \alpha} \text{list}^\alpha \text{int}^C \]

- 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: \(|A^C| = \text{ref} \ |A|\)
- Differentiate between stable and changeable values of a list:
  \(|\text{list}^\alpha \text{int}^C| = \text{list}(\text{int} + \text{ref int})|

\[ \text{map}^{-1} : (\text{ref int} \rightarrow \text{ref int}) \rightarrow \text{list}(\text{int} + \text{ref int}) \rightarrow \text{list}(\text{int} + \text{ref int}) \]
Running Example: \( \text{map} \) (translation II)

\[
\text{case}_{L} e \of
\begin{align*}
\| \emptyset & \rightarrow \emptyset \\
| h :: tl & \rightarrow f h :: \text{map} f tl
\end{align*}
\]
Running Example: map (translation II)

\[\text{map}^\downarrow: (\text{ref int} \rightarrow \text{ref int}) \rightarrow \text{list (int + ref int)} \rightarrow \text{list (int + ref int)}\]

\[
\text{map}^\downarrow f e = \\
\quad \text{caseL } e \text{ of} \\
\quad \quad | [] \rightarrow [] \\
\quad \quad | h :: tl \rightarrow f h :: \text{map} f tl
\]
Running Example: \( \text{map} \) (translation II)

\[
\text{map}:: (\text{ref int} \rightarrow \text{ref int}) \rightarrow \text{list} (\text{int} + \text{ref int}) \rightarrow \text{list} (\text{int} + \text{ref int})
\]

\[
\text{map} f e = \\
\quad \text{caseL} e \text{ of} \\
\quad \mid [] \rightarrow [] \\
\quad \mid h :: tl \rightarrow f h :: \text{map} f tl
\]
Running Example: \(\text{map}\) (translation II)

\[\begin{align*}
\text{map}^{-1} & : (\text{ref int} \to \text{ref int}) \to \text{list} (\text{int} + \text{ref int}) \to \text{list} (\text{int} + \text{ref int}) \\
\text{map} f e & = \\
\quad \text{case} L e \text{ of} \\
\quad \quad | [] \to [] \\
\quad \quad | h :: tl \to f h :: \text{map} f tl \\
\end{align*}\]
Running Example: \( \text{map} \) (translation II)

\[
\text{\textbackslash{\text{map}}} : (\text{ref int} \rightarrow \text{ref int} \rightarrow \text{list (int + ref int)} \rightarrow \text{list (int + ref int)}

\[
\text{map} f e =
\begin{align*}
\text{caseL } e \text{ of} \\
& \text{[]} \rightarrow \text{[]} \\
& h :: tl \rightarrow \text{case } h \text{ of} \\
& \quad h \text{ } \text{ } \rightarrow (\text{inl } ! (f \text{ } \text{ref } h)) :: \text{\textbackslash{\text{map}}} f tl \\
& \quad h_r \rightarrow \text{let } l = \text{ref } ! (f h_r) \text{ in}
\end{align*}
\]
Running Example: map (translation II)

\[
\text{map}^\downarrow : (\text{ref int} \to \text{ref int}) \to \text{list} (\text{int} + \text{ref int}) \to \text{list} (\text{int} + \text{ref int})
\]

\[
\text{map}^\downarrow f e = \\
\begin{align*}
\text{case} \ L e \ of \\
\mid [] & \to [] \\
\mid h :: tl & \to \\
& \text{case} h \ of \\
& \mid h_l & \to (\text{inl} !(f (\text{ref} h_l))) :: \text{map}^\downarrow f tl \\
& \mid h_r & \to \text{let} \ l = \text{ref} !(f \ h_r) \ \text{in} \\
& \quad \text{let} \ () = \\
& \quad \text{push}(l, \lambda().! (f h_r)) \ \text{in}
\end{align*}
\]
Running Example: \( \text{map} \) (translation II)

\[
\text{map} : (\text{ref int} \rightarrow \text{ref int}) \rightarrow \text{list (int + ref int)} \rightarrow \text{list (int + ref int)}
\]

\[
\text{map} f e = \\
\text{case} \_ e \text{ of} \\
| [] \rightarrow [] \\
| h :: tl \rightarrow \\
\text{case} h \text{ of} \\
| h_l \rightarrow (\text{inl }!(f (\text{ref } h_l))) : \text{map} f tl \\
| h_r \rightarrow \text{let } l = \text{ref }!(f h_r) \text{ in} \\
\text{let } () = \\
\text{push}(l, \lambda . !)(f h_r) \text{ in} \\
\text{inr } l : \text{map} f tl
\]
The translation is defined by induction on the typing derivation

Two modes: \( \epsilon = C \) and \( \epsilon = S \)

The code generated in \( C \) mode will be executed from scratch during CP

The code generated in \( S \) mode is self-adjusting

During this mode we record the computations that need to be re-executed during CP

\[ \Delta; \Phi; \Gamma \vdash^\kappa \epsilon : \tau \rightarrow \sqcap e \]
Translation

\[ \Delta; \Phi; \Gamma \vdash^\kappa_e \ e : \tau \hookrightarrow \ l e \]

- The translation is defined by induction on the typing derivation
Translation

\[ \Delta; \Phi; \Gamma \vdash_{\epsilon}^\kappa e : \tau \hookrightarrow \lceil e \rceil \]

- The translation is defined by induction on the typing derivation
- Two modes: \( \epsilon = C \) and \( \epsilon = S \)
The translation is defined by induction on the typing derivation

Two modes: $\epsilon = C$ and $\epsilon = S$

The code generated in $C$ mode will be executed from scratch during CP
The translation is defined by induction on the typing derivation.

Two modes: $\epsilon = C$ and $\epsilon = S$

The code generated in $C$ mode will be executed from scratch during CP.

The code generated in $S$ mode is self-adjusting.

During this mode we record the computations that need to be re-executed during CP.
Change Propagation

\[ Q, \sigma_i[\sigma_c] \sim \sigma_f, c \]
Change Propagation

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

- \( Q \) is the queue holding the recorded computations
Change Propagation

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

- \( Q \) is the queue holding the recorded computations
- While \( Q \) is not empty, the algorithm:
Change Propagation

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

- \( Q \) is the queue holding the recorded computations
- While \( Q \) is not empty, the algorithm:
  - pops an element \((\vec{l}, f)\) from the queue
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:
  - \( \Diamond \) pops an element \((\vec{l}, f)\) from the queue
  - \( \Diamond \) runs the computation \( f() \) that returns the updated values of the locations and incurs cost \( c_f \)
  - \( \Diamond \) 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 \((l, f)\) 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

- \( v_s \) 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.

\[
(3, 42) \approx_{\int}^{\sigma} (3, l)
\]
Soundness, C mode

**Theorem**

Assume that

\[ \vdash^\kappa_C e : \tau \rightarrow \langle e \rangle \]

Then there exist \( v'_s, v'_t, \sigma', j \) and \( c \), such that
Theorem

Assume that

$$\vdash^\kappa_C e : \tau \rightarrow \llbracket e \rrbracket$$

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

$$
\vdash_\mathbb{C}^\kappa e : \tau \rightarrow \llbracket e \rrbracket
$$

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

1. $e \Downarrow v'_s$, $j$
2. $\llbracket e \rrbracket$, $\sigma \Downarrow v'_t$, $\sigma'$, $\emptyset$, $c$
Soundness, $\mathbb{C}$ mode

**Theorem**

Assume that

$$\vdash^\kappa_{\mathbb{C}} e : \tau \rightarrow \lceil e \rceil$$

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

1. $e \downarrow v'_s$, $j$
2. $\lceil e \rceil$, $\sigma \downarrow v'_t$, $\sigma'$, $\emptyset$, $c$
3. $\models c \leq \kappa$
Theorem

Assume that

$$\vdash_{C}^{\kappa} e : \tau \rightarrow \lceil e \rceil$$

Then there exist $v_{s}', v_{t}', \sigma', j$ and $c$, such that

1. $e \Downarrow v_{s}', j$
2. $\lceil e \rceil, \sigma \Downarrow v_{t}', \sigma', \emptyset, c$
3. $\models c \leq \kappa$
4. $v_{s}' \approx_{\sigma}, v_{t}'$
Two-way similarity relation

$$(v_i, v_c) \approx_{(\sigma_i, \sigma_c)}^T v_t$$

- $v_i$ 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 $\sigma_c$

$$\left( (3, 42), (3, 43) \right) \approx_{\text{int}^S \times \text{int}^C}^{[l \mapsto 42], [l \mapsto 43]} (3, l)$$
Soundness, $S$ mode

**Theorem**

Assume that

\[ \cdots; x: \tau' \vdash^\kappa e : \tau \hookrightarrow \Gamma e \downarrow \]

\[ (v_i, v_c) \approx_{(\sigma_i, \sigma_c)} v_t \]

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, $\mathcal{S}$ mode

**Theorem**

Assume that

\[ \vdash_{\mathcal{S}^\kappa} e : \tau \quad \tau \to \overline{e} \]

\[ (v_i, v_c) \approx^\tau (\sigma_i, \sigma_c) v_t \]

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, $\mathcal{S}$ mode

**Theorem**

Assume that

$$\vdash^\kappa_{\mathcal{S}} \cdot; \cdot; x : \tau' \vdash^\kappa e : \tau \hookrightarrow \Gamma e \ \neg

(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) $[x \mapsto v_t] \Gamma e \neg, \sigma_i \Downarrow v'_t, \sigma_f, Q, c$
Soundness, $\mathcal{S}$ mode

**Theorem**

Assume that
\[
\vdash_{\mathcal{S}} \exists x : \tau' \vdash_{\mathcal{S}} e : \tau \hookrightarrow e \Downarrow
\]
\[(v_i, v_c) \approx_{(\sigma_i, \sigma_c)} v_t\]

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. $[x \mapsto v_t]e \Downarrow, \sigma_i \Downarrow v_t', \sigma_f, Q, c$
3. $Q, \sigma_f[\sigma_c] \leadsto \sigma'_f, c'$
Soundness, $\mathbb{S}$ mode

**Theorem**

Assume that

$\cdot; \cdot; x : \tau' \vdash^\kappa_{\mathbb{S}} e : \tau \hookrightarrow \llbracket e \rrbracket$

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

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. $[x \mapsto v_t]\llbracket e \rrbracket, \sigma_i \Downarrow v_t', \sigma_f, Q, c$
3. $Q, \sigma_f[\sigma_c] \leadsto \sigma'_f, c'$
4. $\models c' \leq \kappa$
Soundness, $\mathcal{S}$ mode

**Theorem**

**Assume that**

$$\cdot; \cdot; x : \tau' \vdash^\kappa_{\mathcal{S}} e : \tau \leftarrow \ulcorner e \urcorner$$

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

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. $Q, \sigma_f[\sigma_c] \leadsto \sigma'_f, c'$
4. $\models c' \leq \kappa$
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 proof 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 $C$ and $S$ 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?