Implicit self-adjusting computation for CostIt
Internship Defense

Zoe Paraskevopoulou¹,²
Advisor: Deepak Garg²

¹ENS Cachan  ²Max 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

\[ \frac{8 \times 4 \times 5 + 9 \times 4}{25 \times 5 - 3 \times 2} \]
Change Propagation by Example

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

\[
\begin{align*}
42 & = 17 + 25 \\
42 & = 17 + (5 \times 5) \\
42 & = (8 + 6) + (10 + 4) \\
42 & = (2 \times 4) + (6 + 4) \\
42 & = \frac{3}{20}
\end{align*}
\]
Change Propagation by Example

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

```
+  43  +  25
  +  18  +  10
  +  8  +  6  +  10
  +  2  +  4  +  10  +  3  +  2
  *  8  *  4
  *  2
```

3 / 20
CostI (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:** \( \Delta; \Phi; \Gamma \vdash^\kappa \epsilon; e : \tau \)
  - \( \epsilon \) is the typing mode (S or 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 ($S$ or $C$)
  
  ♦ $\kappa$ is the derived cost

• When $\epsilon = S$ then $\kappa$ is the upper bound of CP
• A type and effect system that allows us to derive upper bounds on the cost of incremental computation

• Judgments: $\Delta; \Phi; \Gamma \vdash_\varepsilon^\kappa e : \tau$
  - $\varepsilon$ is the typing mode ($S$ or $C$)
  - $\kappa$ is the derived cost

• When $\varepsilon = S$ then $\kappa$ is the upper bound of CP

• When $\varepsilon = C$ then $\kappa$ is the worst case execution cost
CostIt (II)

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

$\tau_S$: 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
Costlt (II)

- Functions are annotated with effects $\tau_1^{\mu(\kappa)} \rightarrow \tau_2$
  - 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: $\text{list} [n] \alpha \tau$ - 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 $\leq \kappa$
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
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)
Costlt (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: $\text{map}$ (typing)

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

- If $f$ executes from-scratch with cost $\kappa$ 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: \texttt{map} (typing)

\begin{align*}
\texttt{map}: (\tau_1 \xrightarrow{C(\kappa)} \tau_2) &\xrightarrow{S(0)} \text{list}[n]^\alpha \quad \tau_1 &\xrightarrow{S(\kappa \cdot \alpha)} \text{list}[n]^\alpha \quad \tau_2
\end{align*}

- If \( f \) executes from-scratch with cost \( \kappa \) and \( l \) has \( n \) elements of which at most \( \alpha \) can change then \texttt{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.

\[ \vdash \kappa \epsilon e : \tau \]
\[ \vdash \kappa \epsilon e : \tau \text{ evaluates to } v_1 \]
\[ \vdash \kappa \epsilon e : \tau \approx \]
\[ \vdash \kappa \epsilon e : \tau \text{ evaluates to } v_2 \]
\[ \vdash \kappa \epsilon e : \tau \approx \]
\[ \vdash \kappa \epsilon e : \tau \text{ evaluates to } v_2 \]

(a) First run
(b) Incremental run after input changes

**Figure:** Schematic representation of the basic properties of the translation.
• 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
• 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
• 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: \texttt{map} (translation I)

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

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

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

- 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)

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

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

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

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

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

\[
\text{map}(\text{translation II}) = \text{map} f e = \\
\text{case}_L e \text{ of} \\
| [] \rightarrow [] \\
| h :: tl \rightarrow f h :: \text{map} f \ tl
\]
Running Example: \( \text{map} \) (translation II)

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

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

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

\[
\text{map}^\downarrow f \ e = \\
\text{caseL} \ e \ of \\
\mid [] \rightarrow [] \\
\mid h :: tl \rightarrow f \ h :: \text{map} \ f \ tl
\]
```
map \land f e =
caseL e of
  | [] \rightarrow []
  | h :: tl \rightarrow case h of
    | h_l \rightarrow (inl !(f (ref h_l))) :: \land f tl
    | h_r \rightarrow
```

```
\land: (ref int \rightarrow ref int) \rightarrow list (int + ref int) \rightarrow list (int + ref int)
```
Running Example: \(\text{map} \) (translation II)

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

\[
\text{map} f e = \\
\text{case} \ e \ \text{of} \\
\ | \ [] \ \rightarrow \ [] \\
| \ h :: \ tl \ \rightarrow \ \\
\quad \text{case} \ h \ \text{of} \\
\ | \ h_l \ \rightarrow \ (\text{inl} ! (f (\text{ref} \ h_l))) :: \text{map} f \ tl \\
| \ h_r \ \rightarrow \ \text{let} \ l = \text{ref} ! (f \ h_r) \ \text{in}
\]
Running Example: map (translation II)

```
⌜map⌝: (ref int → ref int) → list (int + ref int) → list (int + ref int)
```

```
map f e =
caseL e of
| [] → []
| h :: tl → f h :: map f tl
```

```
⌜map⌜ f e =
caseL e of
| [] → []
| h :: tl →
case h of
| hL → (inl !(f (ref hL))) ::⌜map⌝ f tl
| hR → let l = ref !(f hR) in
  let () =
    push(l, λ().!(f hR)) in
```

Running Example: map (translation II)

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

\[
\text{map} f e = \\
\text{case}_{\text{L}} e \text{ of} \\
| [] \to [] \\
| h :: tl \to \text{case} h \text{ of} \\
\quad | h_l \to (\text{inl} !(f (\text{ref} h_l))) :: \text{map}^\downarrow f tl \\
\quad | h_r \to \text{let} l = \text{ref} ! (f h_r) \text{ in} \\
\quad \quad \text{let } () = \\
\quad \quad \quad \text{push}(l, \lambda().!(f h_r)) \text{ in} \\
\quad \quad \text{inr} l :: \text{map}^\downarrow f tl
\]
Translation

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

\[ \Delta; \Phi; \Gamma \vdash^\kappa_{\epsilon} e : \tau \rightarrow \llbracket e \rrbracket \]

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

\[ \Delta; \Phi; \Gamma \vdash^\kappa_\epsilon e : \tau \leftrightarrow \overline{\overline{e}} \]

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

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

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

- \( Q \) 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 \((l, f)\) 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 \((\overrightarrow{l}, f)\) 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

\[ v_s \approx_\sigma^\tau v_t \]

- \( v_s \) 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.

\[ (3, 42) \approx_{[l\rightarrow 42]}^{\text{int}^S \times \text{int}^C} (3, l) \]
**Theorem**

Assume that

\[ \vdash^\kappa \mathcal{C} \; e : \tau \rightarrow \lbrack e \rbrack \]

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

1. \( e \Downarrow v'_s, j \)
2. \( \lbrack e \rbrack, \sigma \Downarrow v'_t, \sigma' \)
3. \( |v'_s| = c \)
4. \( v'_s \approx \tau \sigma' v'_t \)
Soundness, $\mathbb{C}$ mode

Theorem

Assume that

\[ \vdash^\kappa e : \tau \rightarrow \boxed{e} \]

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

$$\neg \kappa e : \tau \hookrightarrow \lceil e \rceil$$

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

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

**Theorem**

Assume that

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

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

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

**Theorem**

Assume that

$$\not\models^\kappa_{\mathbb{C}} e : \tau \rightarrow \neg e$$

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

1. $e \downarrow v'_s$, $j$
2. $\neg e \downarrow$, $\sigma \downarrow v'_t$, $\sigma'$, $\emptyset$, $c$
3. $\models c \leq \kappa$
4. $v'_s \approx^\tau_{\sigma^t} v'_t$
Two-way similarity relation

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

- \(v_i\) 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 \(\sigma_c\)

\[
((3, 42), (3, 43)) \approx_{\text{int}^S \times \text{int}^C}^{[l\mapsto 42], [l\mapsto 43]} (3, l)
\]
Theorem

Assume that

\[ \vdash^\kappa_{\mathcal{S}} e : \tau \Rightarrow \preceq e^\top \]

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

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

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

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

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, $S$ mode

**Theorem**

Assume that

$$\cdots; \vdash x : \tau' \vdash^\kappa_S e : \tau \rightarrow \Gamma e \downarrow$$

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

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

**Theorem**

Assume that

\[
\cdot; \cdot; x : \tau' \vdash_{\mathcal{S}}^\kappa e : \tau \rightsquigarrow \lceil e \rceil
\]

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

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]\lceil e \rceil, \sigma_i \Downarrow v'_t, \sigma_f, Q, c
3. Q, \sigma_f[\sigma_c] \rightsquigarrow \sigma'_f, c'
Soundness, $\mathcal{S}$ mode

**Theorem**

Assume that

$$\cdots; x : \tau' \vdash^\kappa_{\mathcal{S}} e : \tau \hookrightarrow \lceil e \rceil$$

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

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. $[x \mapsto v_t]\lceil e \rceil, \sigma_i \downarrow v'_t$, $\sigma_f$, $Q$, $c$
3. $Q, \sigma_f[\sigma_c] \leadsto \sigma'_f$, $c'$
4. $\models c' \leq \kappa$
Soundness, $\mathcal{S}$ mode

**Theorem**

Assume that

$$\vdots; \vdots; x : \tau' \vdash^\kappa_{\mathcal{S}} e : \tau \rightsquigarrow \Gamma e \Downarrow$$

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

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. $[x \mapsto v_t] \Gamma e \Downarrow, \sigma_i \downarrow v'_t, \sigma_f, Q, c$
3. $Q, \sigma_f[\sigma_c] \leadsto \sigma'_f, c'$
4. $| c' \leq \kappa$
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?