Computation Slices as (Universal) Provenance

Umut A. Acar

Max Planck Institute for Software Systems

Joint work with A. Ahmed, J. Cheney, P. Levy, R. Perera

Dagstuhl POP

Provenance in Computation

- Computation is complex mathematical object.
- Treated as a black box: a map from inputs to outputs.
- We need to understand what goes on inside the box.
  Why some output is computed? How it was computed?
  - Debugging: why something went wrong?
  - Security: what kind of data has been used?
  - Reproduciability: can I reproduce the same result?
  - Sensitivity: does the result change when the input changes?

Questions

- What's?
  - What kind of provenance information do we need?
  - What form should the provenance information be?
  - What properties should it have?
- How? How to compute provenance?

Some Examples

Debugging

`mergesort[2,3,1] = [1,3,3]`

What went wrong?

Security

`map increment [6,7,2] = [7,8,3]@(2) = 8`

Have I revealed sensitive information?

Reproducibility

`map increment [6,7,2] = [7, 8 ,3]`

Sensitivity

Would the second element of affected if I change the third element?

`map increment [6,7,2] = [7, 8 ,3]`

How?

Annotation propagation on data

Label individual parts of data and push them through.

- Captures data dependencies.
- Does not capture computational dependencies between code and data.

Consequences: Not useful for debugging, sensitivity analysis, or updates.

Richer annotations

Capture dependencies using a richer language, e.g., some kind of algebra.
Possibly more powerful but still far from Turing complete.

Idea: computation slices

- Reify computations.
- Use computation themselves as forms of provenance.

How?

- Represent computations with expressions that generate them.
- Refine expressions by a form of slicing to generate precise provenance.

Example: Counting Elements

Code

```
fun countElements (xs: α list) =
  case xs of
    nil => 0
  | cons(x,xs) => 1 + countElements xs
```

Run

```
countElements [1,2,3]
```

What is the provenance for full output of the computation?

Example: Counting Elements

Code, Rewritten

```
let countElements =
  fun countElements (xs:  α list) =
    case xs of
      nil => 0
    | cons(x,xs) => 1 + countElements xs
in countElements [1,2,3]
```

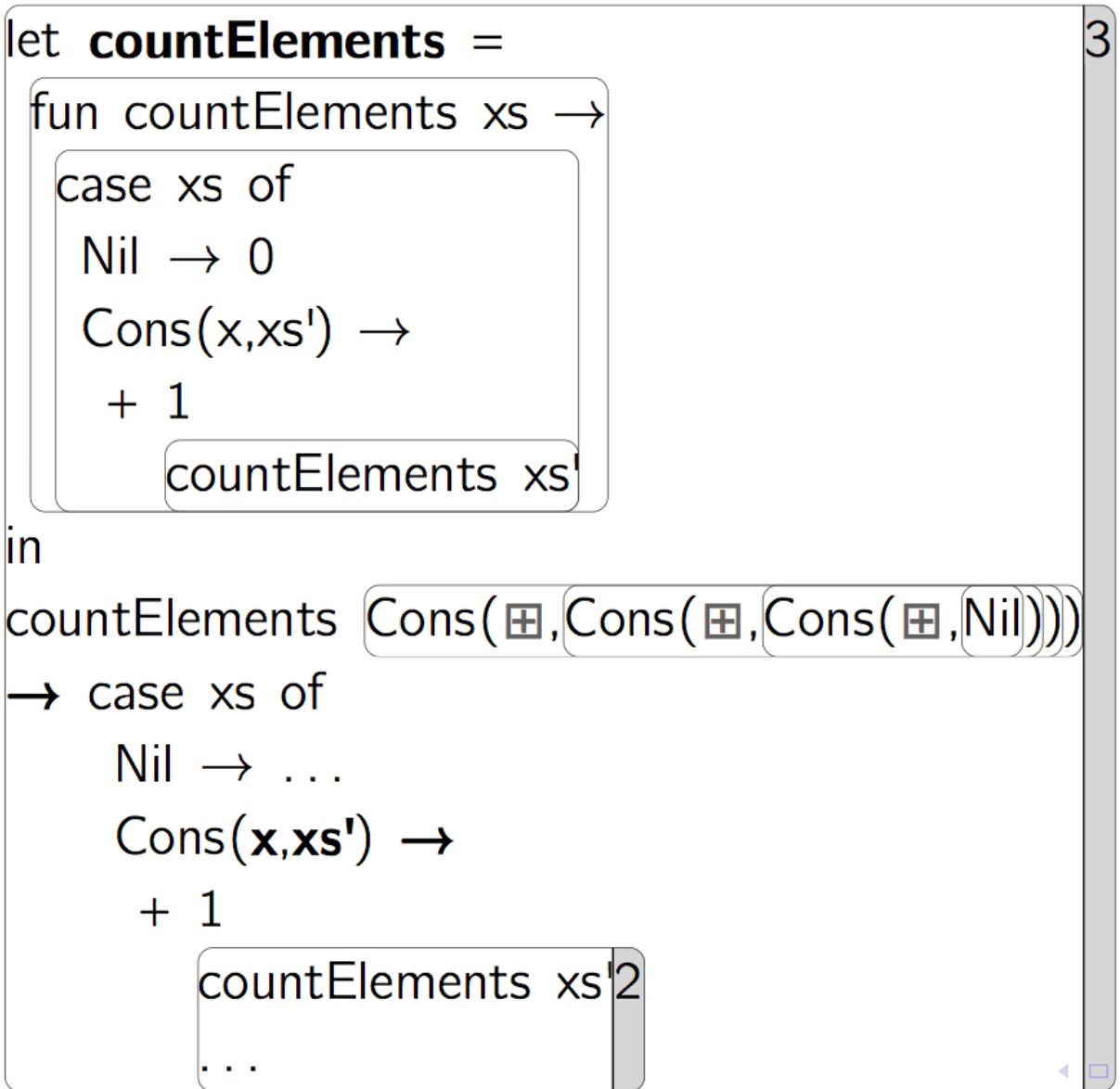
Example: Counting Elements

```
let countElements =                                   3
  fun countElements xs →
    case xs of
      Nil → 0
      Cons(x,xs') →
        + 1
            countElements xs'
in
countElements Cons(⊞,Cons(⊞,Cons(⊞,Nil)))
...
```

Note that the elements of the input are all replaced by a hole.
What does this mean?
It indicates that the output does not depend on the elements at all.

Interactivity

How about the rest of the computation?
Open the box by clicking on the "...."



More complex slicing criteria

Specifying partial data

So far: the output was a simple integer.
What if the output is a more complex data structure, e.g., a list?
Idea: Replace parts of the output of no interest with holes $\square$.

Examples

Don't care about

- the whole output: $\square$
- the second element: [1, $\square$, 3]
  Cons(1, Cons($\square$, Cons (3, Nil)))
- the whole tail after the second element: Cons(1, Cons(2, $\square$))
- a tree except for a path: Node($\square$,1, Node($\square$,2,Leaf))

Example: Map

Code for map

```
fun map f (xs: α list) =
  case xs of
    nil => nil
  | cons(x,xs') => Cons (f x, map f xs')
```

Example: Map

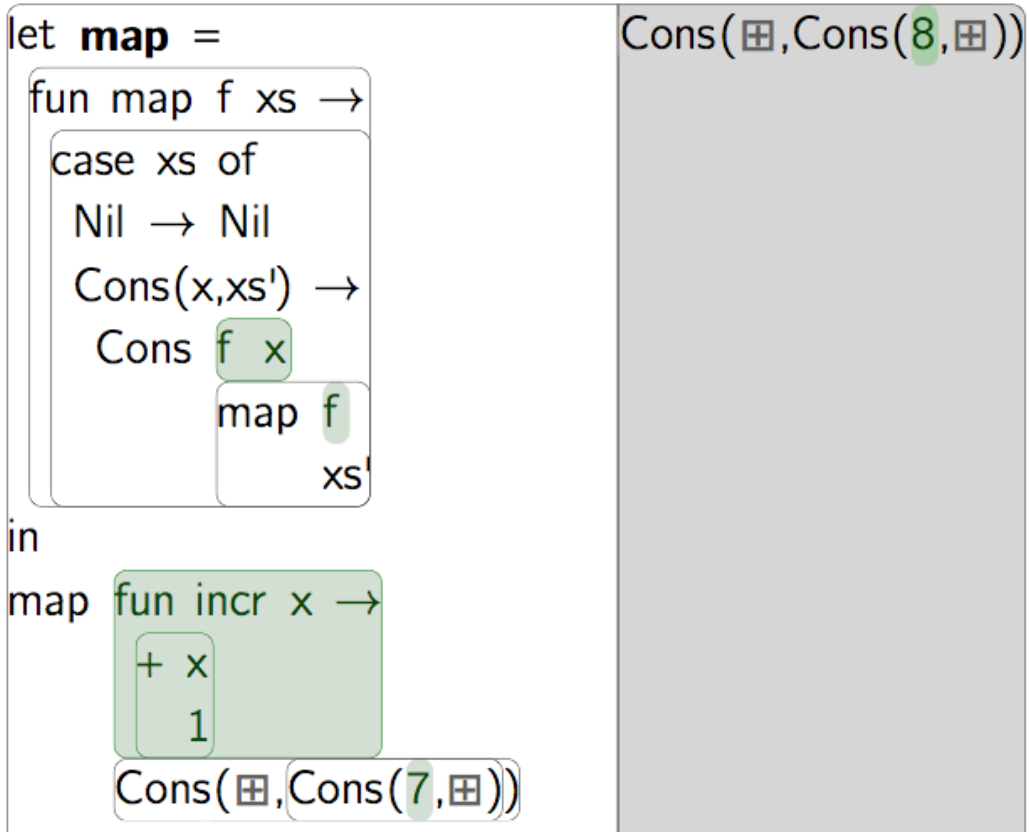
Code, Rewritten

```
let map =
  fun map f (xs:  α list) =
    case xs of
      nil => nil
    | cons(x,xs') => Cons (f x, map f xs')
in map (fun incr x → x + 1) [6,7,2]
```

Provenance question

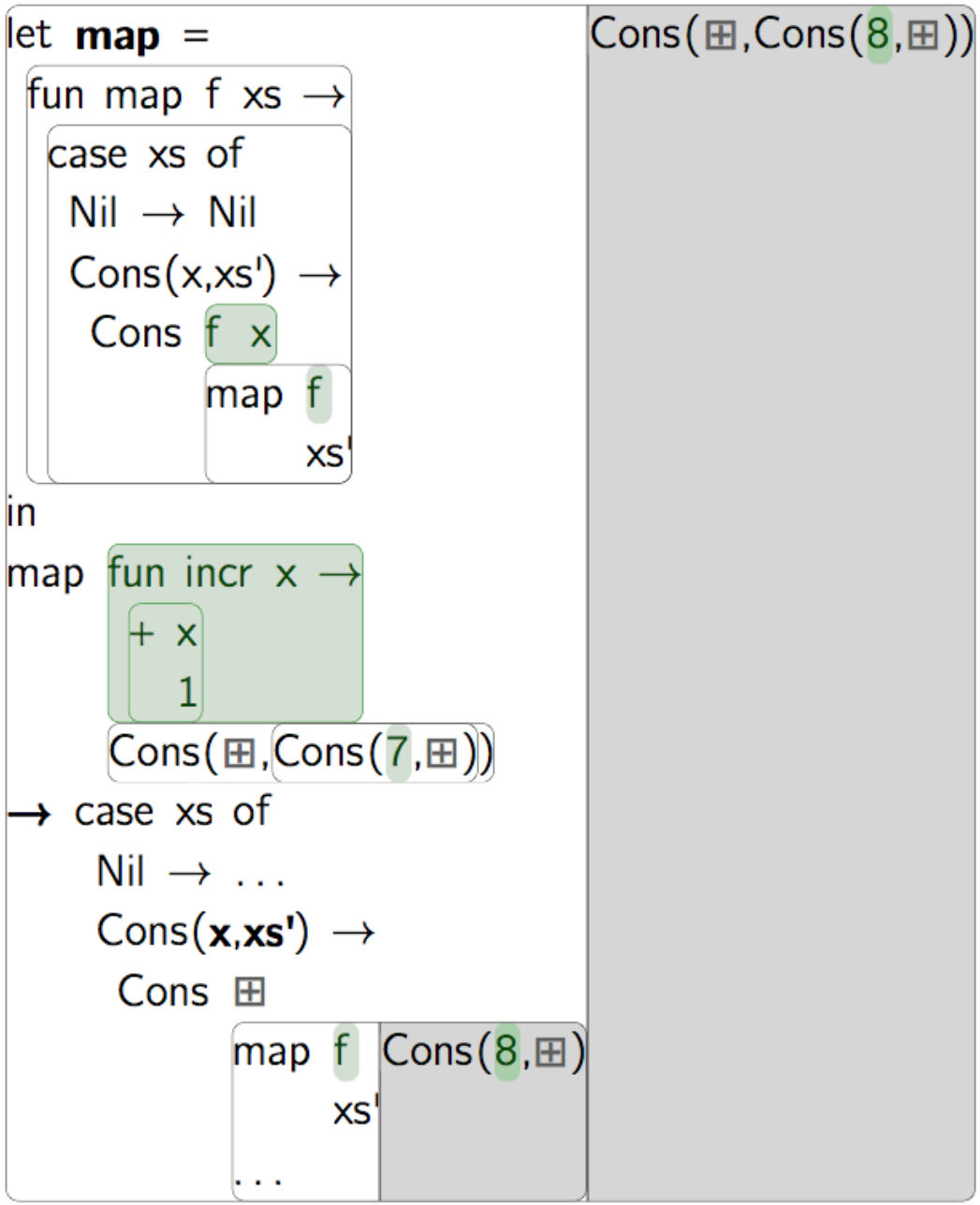
Compute a slice for the second element of the output.
Partial value: [□,8,□]

Map: Program Since



Note: parts of the input are replaced with holes.

Map: Computation Slice



Focus Slice

Partial values

Partial values: replace any part of a data with a hole.
Allows us to eliminate part of the value we don't care about.

Partial values with focus

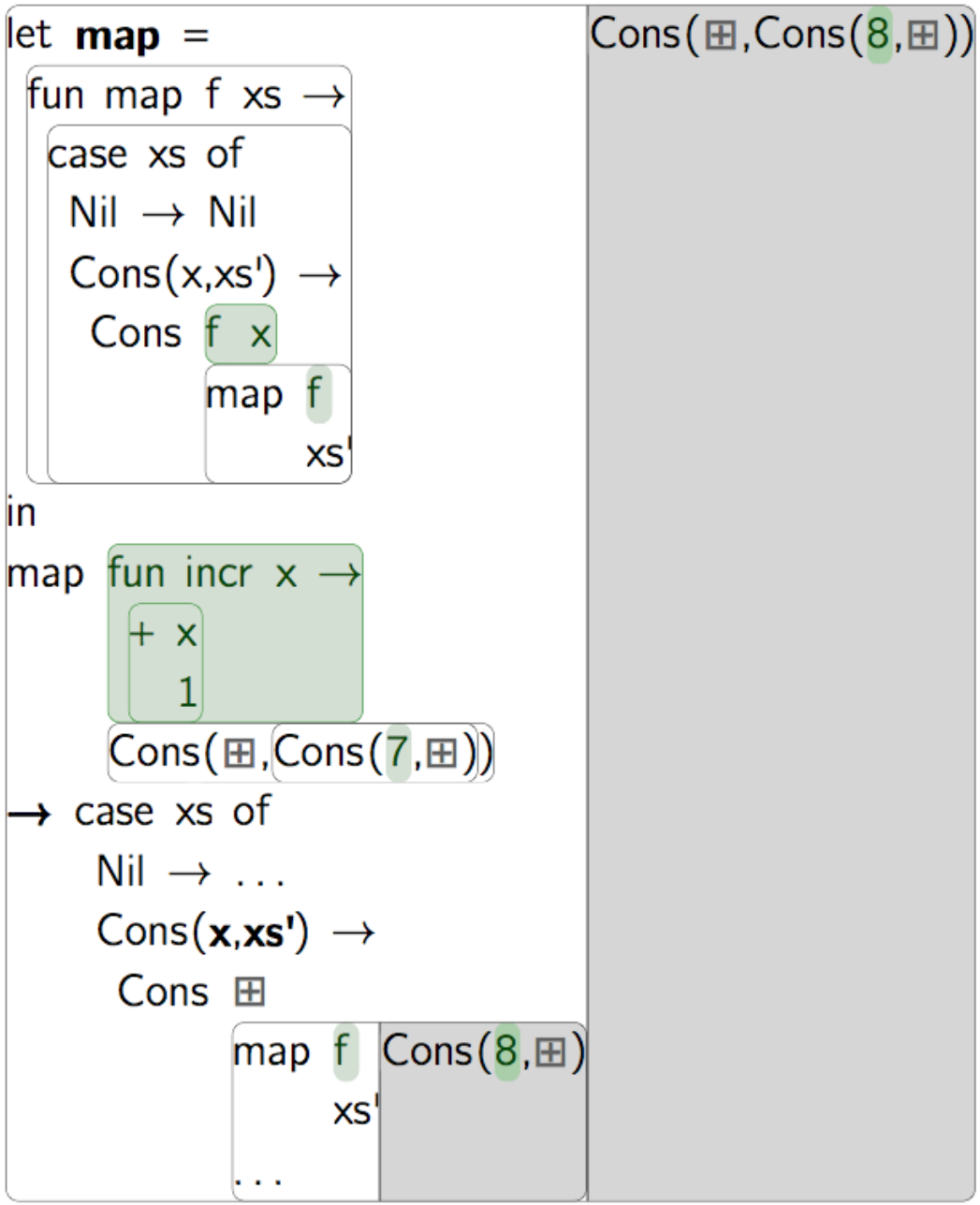
Focus: allow focusing on parts of a partial value.
Show the provenance for focused bit of the partial value.

Partial value with and without focus covers all range of queries.
You can drop or focus on any part of some data.

Examples

- Focus on the full result: 3
- Focus on second: [7, 8 ,3]
- Focus on second and throw away first and third: [□, 8 ,□]

Map: Computation Slice



Example: Mergesort

Buggy merge

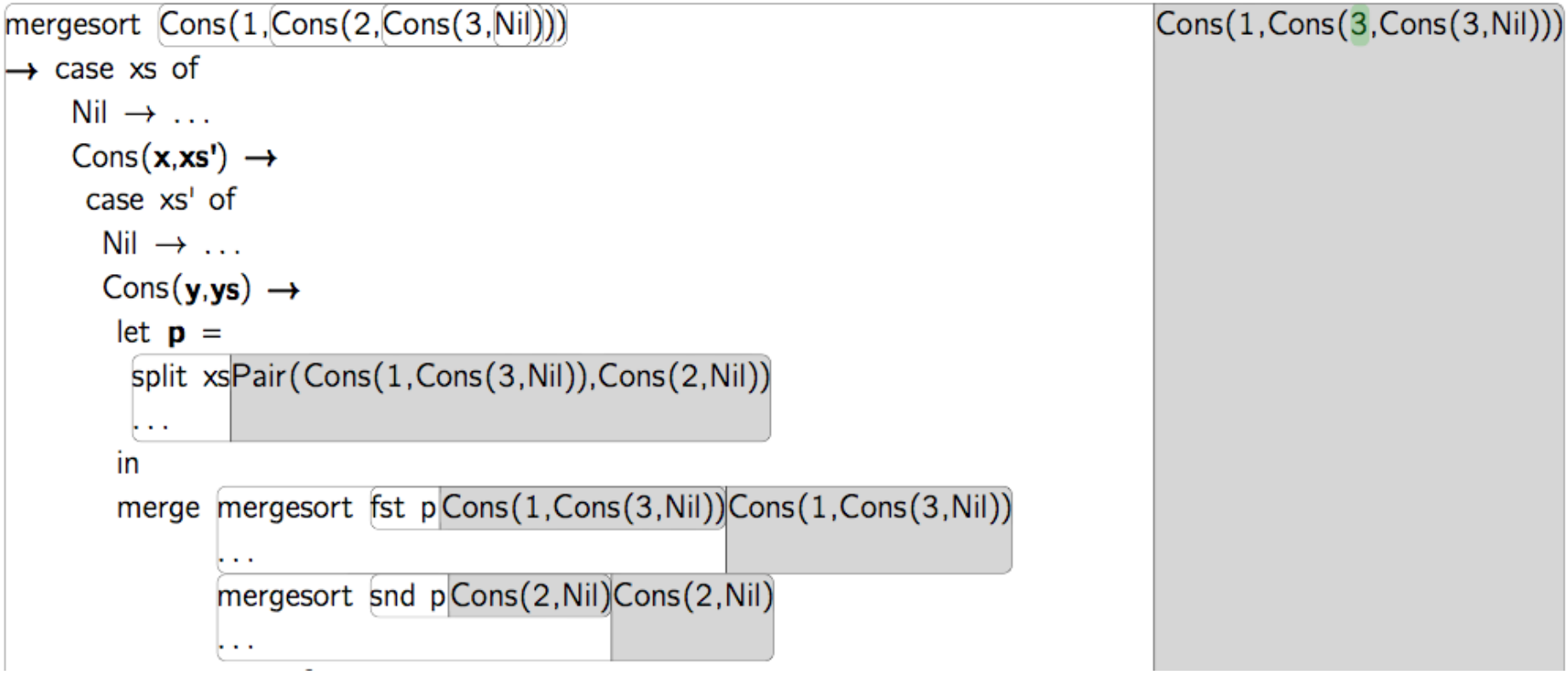
```
fun merge (xs:  int list, ys:  int list) =
  case xs of
    nil => ys
  | cons(x,xs') => case ys of
                     nil => xs
                   | cons(y,ys') =>
                       case (x < y) of
                         True => cons (x, merge (xs', ys)
                       | False => cons (x, merge (xs, ys')
```

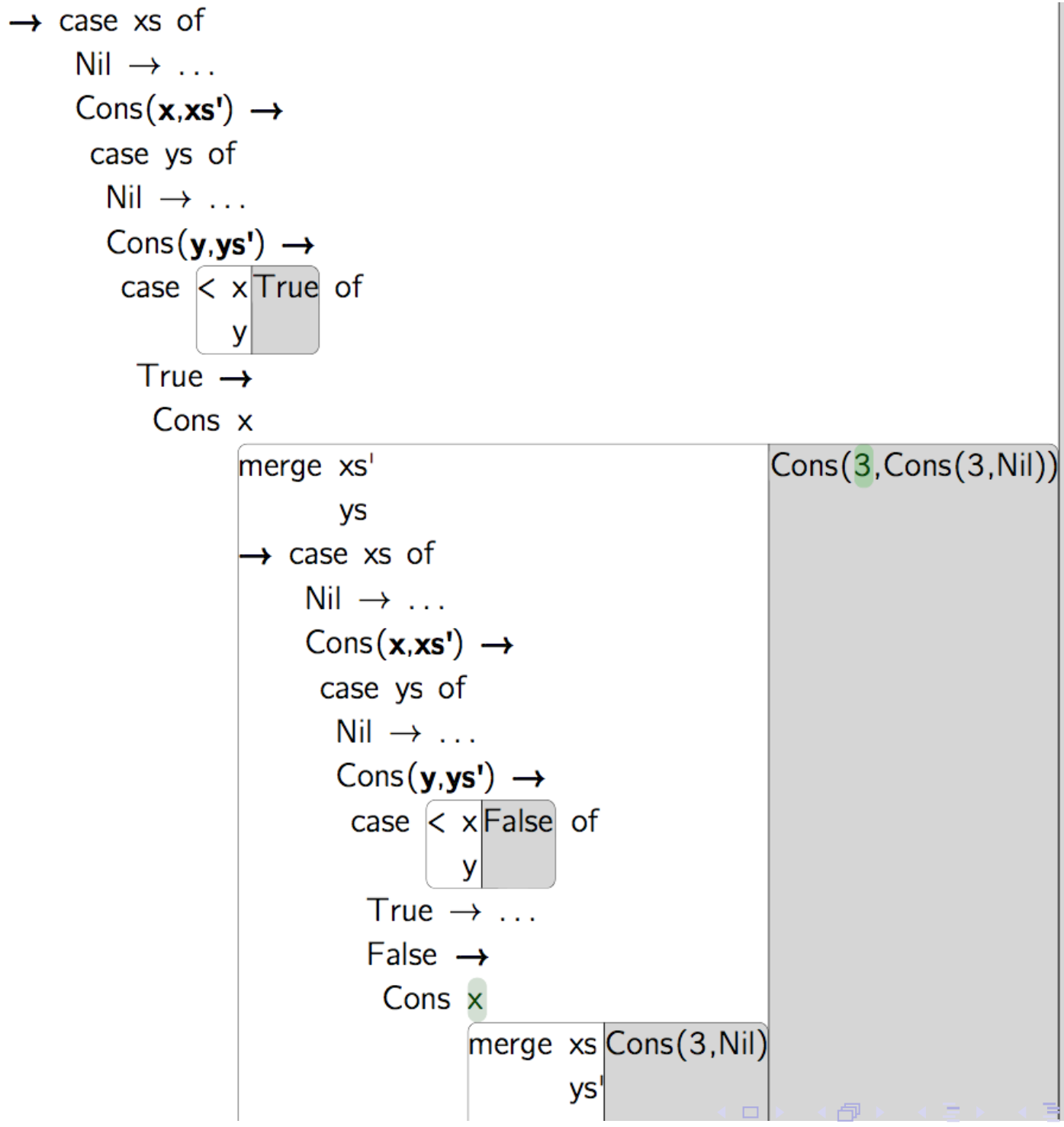
Run mergesort

mergesort[1,2,3] = [1,3,3]

Mergesort Slice

Slice with respect to a partial result focusing on the second element.




→ case xs of
Nil → ...
Cons(x,xs') →
case ys of
Nil → ...
Cons(y,ys') →
case < x y True of
True →
Cons x
merge xs' ys
Cons(3,Cons(3,Nil))
→ case xs of
Nil → ...
Cons(x,xs') →
case ys of
Nil → ...
Cons(y,ys') →
case < x y False of
True → ...
False →
Cons x
merge xs ys' Cons(3,Nil)

The Language

A general-purpose language

Lambda calculus, extended with

- products, sums, and recursive types,
- recursive functions.

Types	τ	$::=$	$\mathbf{1} \mid b \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \mu\alpha.\tau \mid \alpha$
Contexts	Γ	$::=$	$\bullet \mid \Gamma, x : \tau$
Expressions	e	$::=$	$() \mid c \mid \oplus(\vec{e}) \mid \mathbf{fun}\ f(x).e \mid x \mid e_1\ e_2 \mid (e_1, e_2)$
		$\mid$	$\mathbf{fst}\ e \mid \mathbf{snd}\ e \mid \mathbf{case}\ e\ \mathbf{of}\ \{\mathbf{inl}(x_1).e_1; \mathbf{inr}(x_2).e_2\}$
		$\mid$	$\mathbf{inl}\ e \mid \mathbf{inr}\ e \mid \mathbf{roll}\ e \mid \mathbf{unroll}\ e$
Values	v	$::=$	$c \mid (v_1, v_2) \mid \mathbf{inl}\ v \mid \mathbf{inr}\ v \mid \mathbf{roll}\ v$
		$\mid$	$\langle \rho, \mathbf{fun}\ f(x).e \rangle$
Environments	ρ	$::=$	$\bullet \mid \rho[x \mapsto v]$

Tracing Evaluation

Ideas

- Trace evaluations (executions).
- Structure the trace to be symmetric to expressions.

Standard evaluation: $\rho, e \Downarrow v$
Tracing evaluation: $\rho, e \Downarrow v, T$
Evaluating traces: $\rho, T \Downarrow v$

$$
\begin{array}{rrcl}
\text{Traces} & T & ::= & c \mid x \mid \oplus_{\vec{c}}(\vec{T}) \mid \textbf{let } x = T_1 \textbf{ in } T_2 \\
& & \mid & (T_1, T_2) \mid \textbf{fst } T \mid \textbf{snd } T \mid \textbf{inl } T \mid \textbf{inr } T \\
& & \mid & \textbf{roll } T \mid \textbf{unroll } T \mid \textbf{fun } f(x).e \\
& & \mid & \textbf{case } T \textbf{ of } \{\textbf{inl}(x_1).T_1; \textbf{inr}(x_2).e_2\} \\
& & \mid & \textbf{case } T \textbf{ of } \{\textbf{inl}(x_1).e_1; \textbf{inr}(x_2).T_2\} \\
& & \mid & T_1\ T_2 \rhd f(x).T
\end{array}
$$

$\boxed{\rho, e \Downarrow v, T}$

$$\frac{}{\rho, x \Downarrow \rho(x), x} \qquad \frac{}{\rho, c \Downarrow c, c}$$

$$\frac{}{\rho, \mathbf{fun}\ f(x).e \Downarrow \langle \rho, \mathbf{fun}\ f(x).e \rangle, \mathbf{fun}\ f(x).\tau e}$$

$$\frac{\rho, e_1 \Downarrow v_1, T_1 \quad \rho, e_2 \Downarrow v_2, T_2 \quad \rho[f \mapsto v_1][x \mapsto v_2], e \Downarrow v, T}{\rho, e_1\ e_2 \Downarrow v, T_1\ T_2 \rhd f(x).T}\ v_1 = \langle \rho, \mathbf{fun}\ f(x).e \rangle$$

$$\frac{\rho, e_1 \Downarrow v_1, T_1 \quad \rho, e_2 \Downarrow v_2, T_2}{\rho, (e_1, e_2) \Downarrow (v_1, v_2), (T_1, T_2)} \qquad \frac{\rho, e \Downarrow (v_1, v_2), T}{\rho, \mathbf{fst}\ e \Downarrow v_1, \mathbf{fst}\ T}$$

$$\frac{\rho, e \Downarrow v, T}{\rho, \mathbf{inl}\ e \Downarrow \mathbf{inl}\ v, \mathbf{inl}\ T} \qquad \frac{\rho, e \Downarrow \mathbf{inl}\ v_1, T \quad \rho[x_1 \mapsto v_1], e_1 \Downarrow v, T_1}{\begin{array}{c}\rho, \mathbf{case}\ e\ \mathbf{of}\ \{\mathbf{inl}(x_1).e_1; \mathbf{inr}(x_2).e_2\} \Downarrow \\ v, \mathbf{case}\ T\ \mathbf{of}\ \{\mathbf{inl}(x_1).T_1; \mathbf{inr}(x_2).e_2\}\end{array}}$$

Punching holes

$$
\begin{array}{lrcl}
\text{Expressions} & e & ::= & \ldots \mid \Box \\
\text{Values} & v & ::= & \ldots \mid \Box \\
\text{Traces} & T & ::= & \ldots \mid \Box \mid T_1\ e_2 \\
 & & \mid & \textbf{case}\ T\ \textbf{of}\ \{\textbf{inl}(x_1).e_1; \textbf{inr}(x_2).e_2\}
\end{array}
$$

$\boxed{\rho, e \Downarrow v, T}$

$$
\ldots \qquad \frac{}{\rho, \Box \Downarrow \Box, \Box} \qquad \frac{\rho, e_1 \Downarrow \Box, T_1}{\rho, e_1\ e_2 \Downarrow \Box, T_1\ e_2} \qquad \frac{\rho, e \Downarrow \Box, T}{\rho, \textbf{fst}\ e \Downarrow \Box, \textbf{fst}\ T}
$$

$$
\frac{\rho, e \Downarrow \Box, T}{\rho, \textbf{snd}\ e \Downarrow \Box, \textbf{snd}\ T} \qquad \frac{\rho, e \Downarrow \Box, T}{\begin{array}{c}\rho, \textbf{case}\ e\ \textbf{of}\ \{\textbf{inl}(x_1).e_1; \textbf{inr}(x_2).e_2\} \Downarrow \\ \Box, \textbf{case}\ T\ \textbf{of}\ \{\textbf{inl}(x_1).e_1; \textbf{inr}(x_2).e_2\}\end{array}}
$$

$$
\frac{\rho, e \Downarrow \Box, T}{\rho, \textbf{unroll}\ e \Downarrow \Box, \textbf{unroll}\ T}
$$

Unevaluating Partial Traces to Expressions

$$\boxed{v, T \Downarrow^{-1} \rho, e} \qquad \frac{}{\Box, T \Downarrow^{-1} \Box, \Box} \qquad \frac{}{c, c \Downarrow^{-1} \Box, c}$$

$$\frac{}{\langle \rho, \textbf{fun}\ f(x).e \rangle, \textbf{fun}\ f(x).e' \Downarrow^{-1} \rho, \textbf{fun}\ f(x).e}$$

$$\frac{\begin{array}{c} v, T \Downarrow^{-1} \rho[f \mapsto v_1][x \mapsto v_2], e \\ v_2, T_2 \Downarrow^{-1} \rho_2, e_2 \qquad v_1 \sqcup \langle \rho, \textbf{fun}\ f(x).e \rangle, T_1 \Downarrow^{-1} \rho_1, e_1 \end{array}}{v, T_1\ T_2 \rhd f(x).T \Downarrow^{-1} \rho_1 \sqcup \rho_2, e_1\ e_2}\ v \neq \Box$$

$$\frac{v_2, T_2 \Downarrow^{-1} \rho_2, e_2 \qquad v_1, T_1 \Downarrow^{-1} \rho_1, e_1}{(v_1, v_2), (T_1, T_2) \Downarrow^{-1} \rho_1 \sqcup \rho_2, (e_1, e_2)} \qquad \frac{(v_1, \Box), T \Downarrow^{-1} \rho, e}{v_1, \textbf{fst}\ T \Downarrow^{-1} \rho, \textbf{fst}\ e}\ v_1 \neq \Box$$

$$\frac{v, T_1 \Downarrow^{-1} \rho[x_1 \mapsto v_1], e_1 \qquad \textbf{inl}\ v_1, T \Downarrow^{-1} \rho, e}{\begin{array}{c} v, \textbf{case}\ T\ \textbf{of}\ \{\textbf{inl}(x_1).T_1; \textbf{inr}(x_2).e_2\} \Downarrow^{-1} \\ \rho, \textbf{case}\ e\ \textbf{of}\ \{\textbf{inl}(x_1).e_1; \textbf{inr}(x_2).\Box\} \end{array}}\ v \neq \Box$$

Slicing Traces

Definition (Trace Slice)

Let T be a consistent trace $\rho, T \Downarrow v$. We define a *(trace) slice* for a pattern $u \sqsubseteq v$ to be any $(\rho', S) \sqsubseteq (\rho, T)$ such that $\rho', S \Downarrow v', S'$ with $v' \sqsupseteq u$ and $S' \sqsupseteq S$.

Recall: traces and expressions are identical.
We can reuse almost all the rules, except for the function application.

$\boxed{p, T \searrow \rho, S}$

$$\frac{p, T \Downarrow^{-1} \_, e \qquad p, T \searrow \rho[f \mapsto p_1][x \mapsto p_2], S \quad p_1 \sqcup \langle \rho, \textbf{fun}\ f(x).e \rangle, T_1 \searrow \rho_1, S_1 \qquad p_2, T_2 \searrow \rho_2, S_2}{p, T_1\ T_2 \rhd f(x).T \searrow \rho_1 \sqcup \rho_2, S_1\ S_2 \rhd f(x).S}$$

Theorems

- **Determinism:** If $v, T \Downarrow^{-1} \rho, e$ and $v, T \Downarrow^{-1} \rho', e'$ then $(\rho, e) = (\rho', e')$.
- **Totality:** If $u \sqsubseteq v$, there unique ρ', e' such that $u, T \Downarrow^{-1} \rho', e'$. We write (ρ', e') for $\Downarrow^{-1}(u, T)$.
- **Monotonicity:** If $u \sqsubseteq u' \sqsubseteq v$ then $\Downarrow^{-1}(u, T) \sqsubseteq \Downarrow^{-1}(u', T)$.
- **Consistency:** $\Downarrow^{-1}(v, T) \sqsubseteq (\rho, e)$.

Theorem (Correctness of trace slicing)

If $\rho, T \Downarrow v$ and $u \sqsubseteq v$ then $u, T \searrow \rho', S$ with $(\rho', S) \sqsubseteq (\rho, T)$. Moreover, (ρ', S) is a slice of (ρ, T) for u.

Theorem (Minimality of slices)

Let $\rho, T \Downarrow v, T$ and $u \sqsubseteq v$ be a partial value s.t. $u, T \searrow \rho', S$. For any trace slice $(S', \rho'') \sqsubseteq (T, \rho)$ for u, we have $(S, \rho') \sqsubseteq (S', \rho'')$.

Conclusion

Interested in provenance techniques for general-purpose languages.
Derived provenance should have the following properties.

- Executable.
- Completeness: must not omit any important information
- Precision: must not contain more than necessary, be minimal.

Ideas

This seems possible.

- Trace executions
- Structure traces in the same way as expressions.
  Traces and expressions are isomorphic.
- Expressions and trace with holes and focus: allow expression tracing criteria as well as provenance.

Thank you!
Questions?