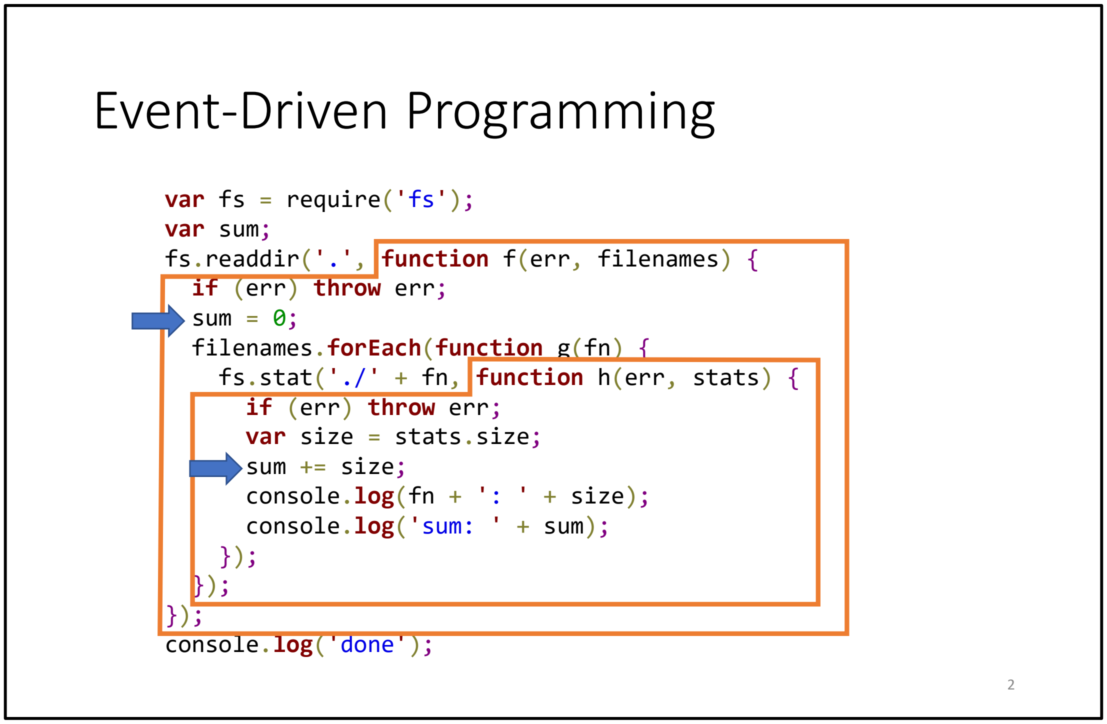
Precise Dataflow Analysis of Event-Driven Applications

Ming-Ho Yee, Ayaz Badouraly, Ondřej Lhoták, Frank Tip, Jan Vitek

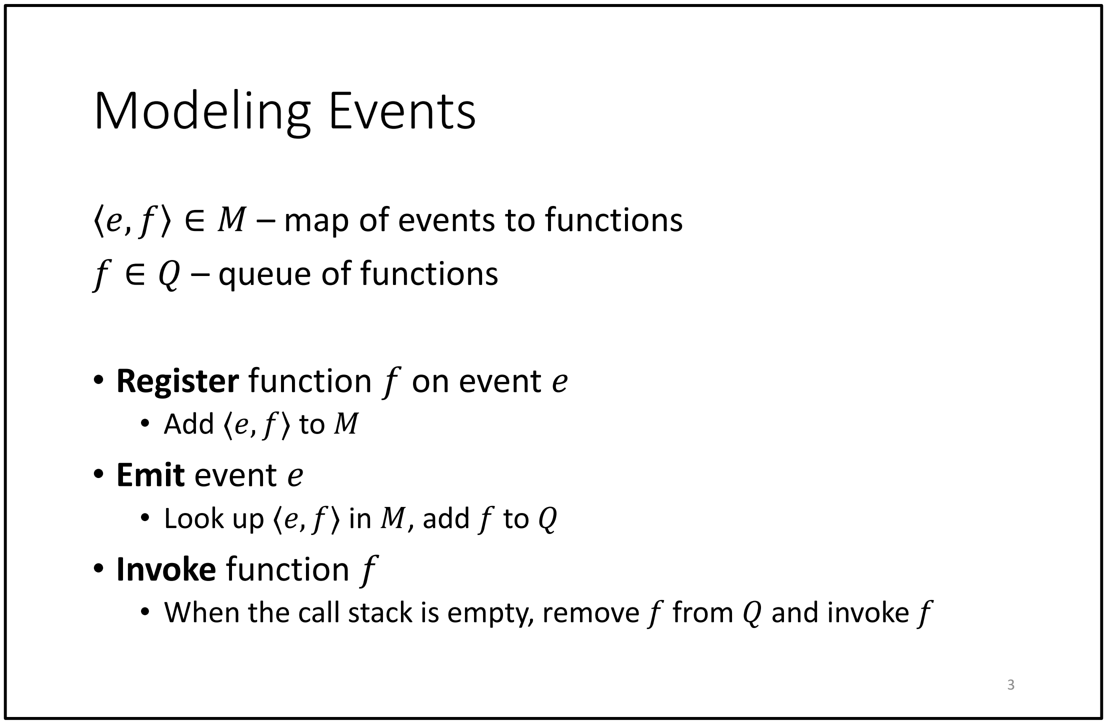
January 23, 2020



Event-Driven Programming

```
var fs = require('fs');
var sum;
fs.readdir('.', function f(err, filenames) {
  if (err) throw err;
  sum = 0;
  filenames.forEach(function g(fn) {
    fs.stat('./' + fn, function h(err, stats) {
      if (err) throw err;
      var size = stats.size;
      sum += size;
      console.log(fn + ': ' + size);
      console.log('sum: ' + sum);
    });
  });
});
console.log('done');
```

- This is an example of event-driven programming
  - Functions ‘f’ and ‘h’ are registered as callbacks
    - ‘g’ is synchronous
  - When the program runs, the callbacks are registered but execute later
  - ‘done’ is printed
  - When ‘readdir’ returns, its result is an array of filenames which is passed into ‘f’
  - For each filename, call ‘stat’ and register a callback
  - Sum up the file sizes
- Now consider doing a static analysis
  - The analysis does not know what order the callbacks execute in
  - So the analysis observes that ‘sum’ could be read before it is written to
  - Therefore, “bug,” even though this never occurs in a concrete execution
- Goal: build a static analysis that accounts for the order of event handler execution



Modeling Events

$\langle e, f \rangle \in M$ – map of events to functions

$f \in Q$ – queue of functions

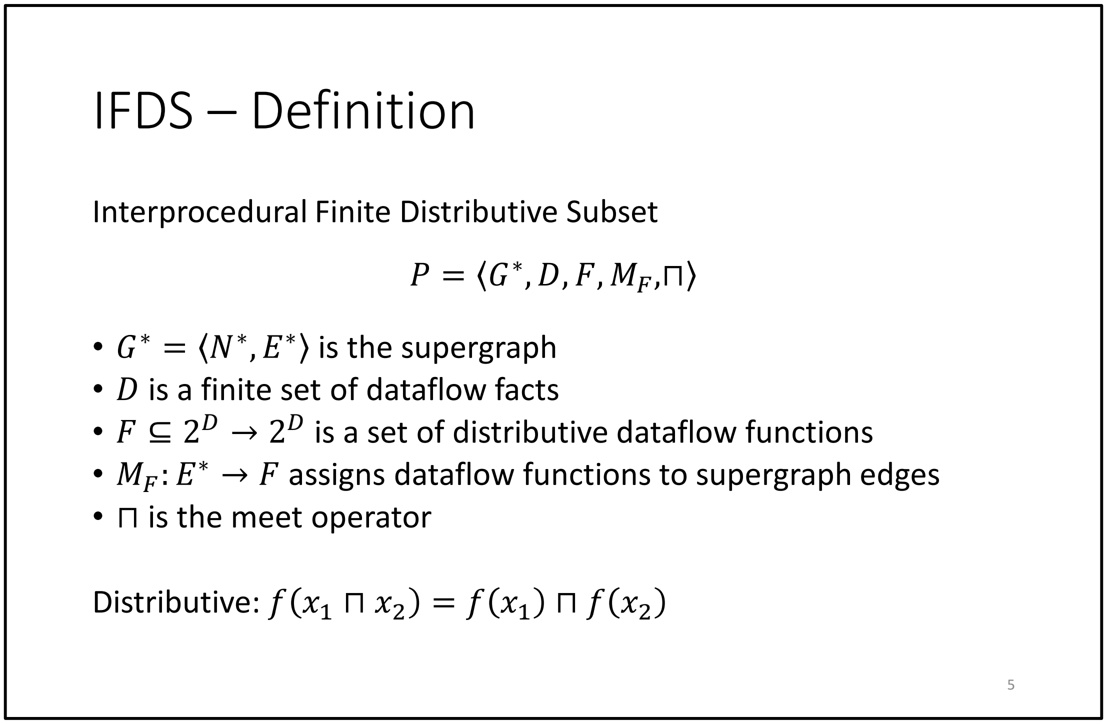
- **Register** function f on event e
  - Add $\langle e, f \rangle$ to M
- **Emit** event e
  - Look up $\langle e, f \rangle$ in M, add f to Q
- **Invoke** function f
  - When the call stack is empty, remove f from Q and invoke f

- Program maintains a map M of events to functions, and a queue Q of functions
- There are three operations: register, emit, and invoke
  - Register function f on event e: add the pair <e, f> to M
  - Emit event e: look up the pair <e, f> in M and add f to Q
  - Invoke function f
    - When the top-level function finishes, the call stack is empty
    - Continuously remove a function f from Q, and invoke f
    - This may register additional event handlers and/or emit additional events
    - Execution terminates when Q is empty

IFDS and IDE Frameworks

6:00 mark

- We take a given IFDS analysis and augment it with information about the event handler ordering
  - We do so by transforming to the IDE framework, which generalizes IFDS



IFDS – Definition

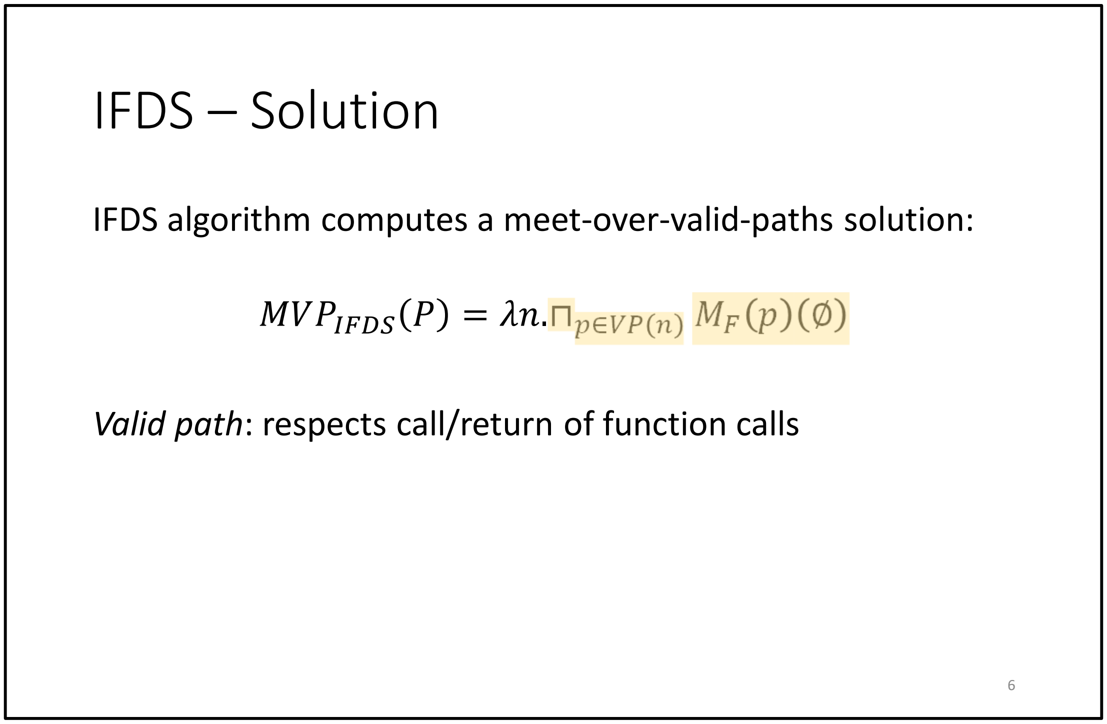
Interprocedural Finite Distributive Subset

$$P = \langle G^*, D, F, M_F, \sqcap \rangle$$

- $G^* = \langle N^*, E^* \rangle$ is the supergraph
- D is a finite set of dataflow facts
- $F \subseteq 2^D \rightarrow 2^D$ is a set of distributive dataflow functions
- $M_F: E^* \rightarrow F$ assigns dataflow functions to supergraph edges
- $\sqcap$ is the meet operator

Distributive: $f(x_1 \sqcap x_2) = f(x_1) \sqcap f(x_2)$

- *Interprocedural* analysis, computes *subset* of a *finite* set, and dataflow functions are *distributive*
- An instance is described as a 5-tuple
    - You must provide the program to be analyzed, and the specification of a dataflow analysis
    - G* is the interprocedural control-flow graph, also called a supergraph
    - D is a finite set of dataflow facts, e.g. live variables, uninitialized variables, busy expressions, that the analysis computes
    - F is a set of distributive functions that describe how the dataflow facts are updated
    - M assigns the dataflow functions to edges of the supergraph
    - Meet is how to merge the information from two separate branches
- Distributive is the key requirement: can compute the result by looking at the input set element-by-element
    - I.e., only need to look at one element of the input at a time



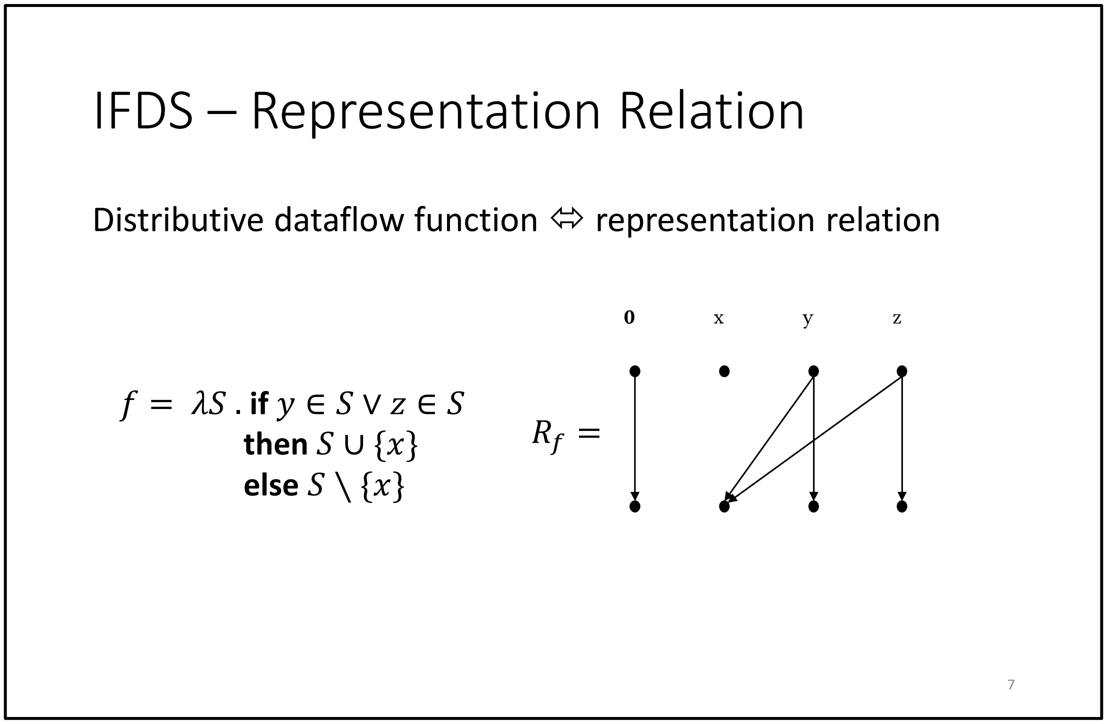
IFDS – Solution

IFDS algorithm computes a meet-over-valid-paths solution:

$$MVP_{IFDS}(P) = \lambda n. \sqcap_{p \in VP(n)} M_F(p)(\emptyset)$$

Valid path: respects call/return of function calls

- Solution is called the “meet-over-valid-paths”
    - A *valid path* means a function returns to its call site and not some other call site
- For a given program node ‘n’, compute all the valid paths ‘p’ from start of the program to n
    - Compose the transfer functions along that path ‘p’
    - Apply the empty set (initial value)
    - This computes the dataflow result for a particular path ‘p’
    - Then take the meet over all those paths to get a combined answer

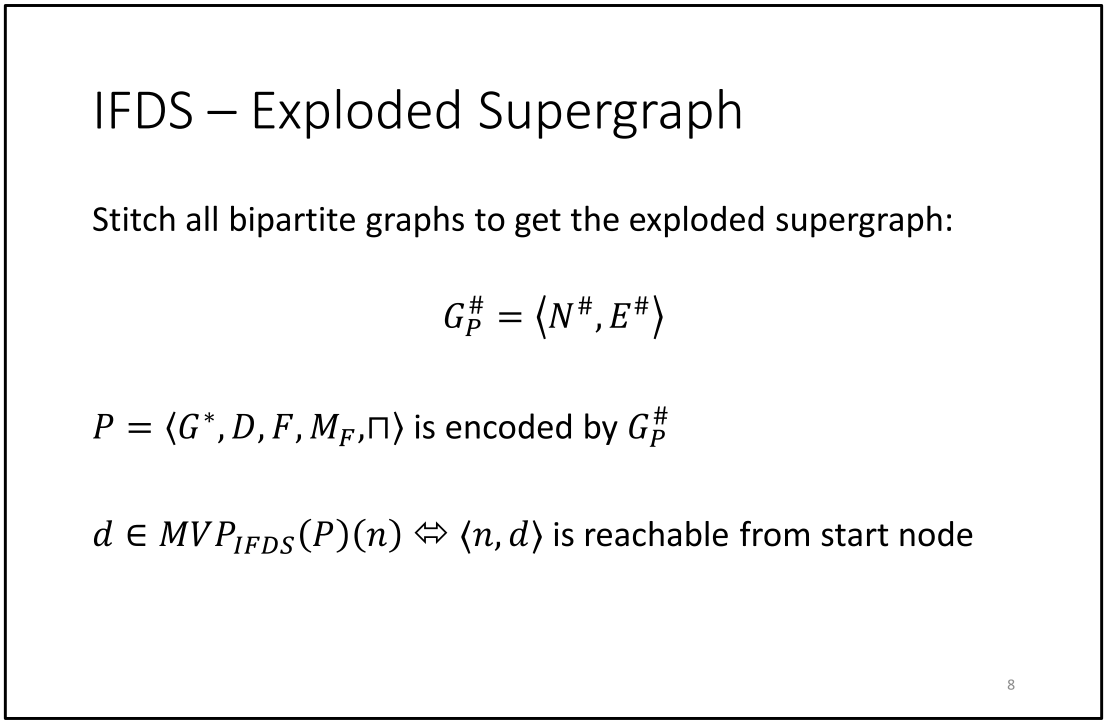


IFDS – Representation Relation

Distributive dataflow function ⇔ representation relation

$$
\begin{aligned}
f = \lambda S\,.\,&\textbf{if } y \in S \vee z \in S\\
&\textbf{then } S \cup \{x\}\\
&\textbf{else } S \setminus \{x\}
\end{aligned}
\qquad R_f =
$$

- Key: every distributive dataflow function has a *representation relation*
    - i.e., a bipartite graph
- Need a “zero” node (roughly corresponding to the empty set), plus nodes for each element in the set D
- Distributive: based on only one input element, what is the output?
    - Then merge the inputs
- Exact details not super important here



IFDS – Exploded Supergraph

Stitch all bipartite graphs to get the exploded supergraph:

$$G_P^{\#} = \left\langle N^{\#}, E^{\#} \right\rangle$$

$P = \langle G^*, D, F, M_F, \sqcap \rangle$ is encoded by $G_P^{\#}$

$d \in MVP_{IFDS}(P)(n) \Leftrightarrow \langle n, d \rangle$ is reachable from start node

- Each edge in the supergraph has a dataflow function
  - Therefore each edge in the supergraph has a representation relation
  - Stitch all these relations together to get the exploded supergraph
- The exploded supergraph encodes an IFDS problem instance, i.e. both the program to be analyzed and the dataflow analysis
- Transform the dataflow analysis into a graph reachability problem
  - d is a dataflow fact for node n if <n,d> is reachable

IDE – Generalization of IFDS

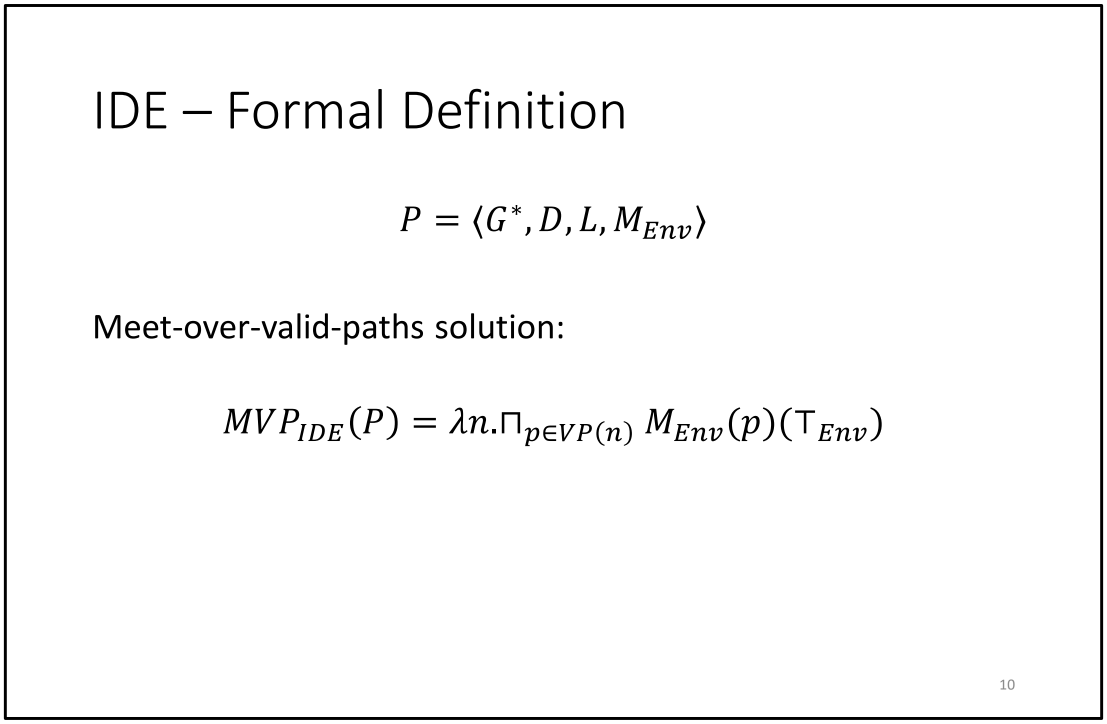
Interprocedural Distributive Environment

L is a finite-height lattice used for the analysis

- Environment $D \rightarrow L$
  - Dataflow set D
- Distributive environment transformer $(D \rightarrow L) \rightarrow (D \rightarrow L)$
  - Distributive dataflow function $D \rightarrow D$

12:00 (+6:00) mark

- IDE is a generalization of the IFDS framework
- First, we need a lattice L
  - A set with a partial order, least upper bound, and greatest lower bound
    - Given two elements, can find a lub or glb that “captures” the information in the two elements
  - This is used for the static analysis, so it needs to have finite height
  - As the analysis runs, it computes values in the lattice
  - Values can only go in one direction (in this case, down the lattice), so termination is guaranteed
- IDE computes environments in D to L
  - Generalization of the dataflow set D in IFDS
  - Instead of computing elements of a set, compute values associated with those elements
- Update environments with environment transformers
  - Generalization of the dataflow functions in IFDS
  - Environment transformers attached to each edge of the graph



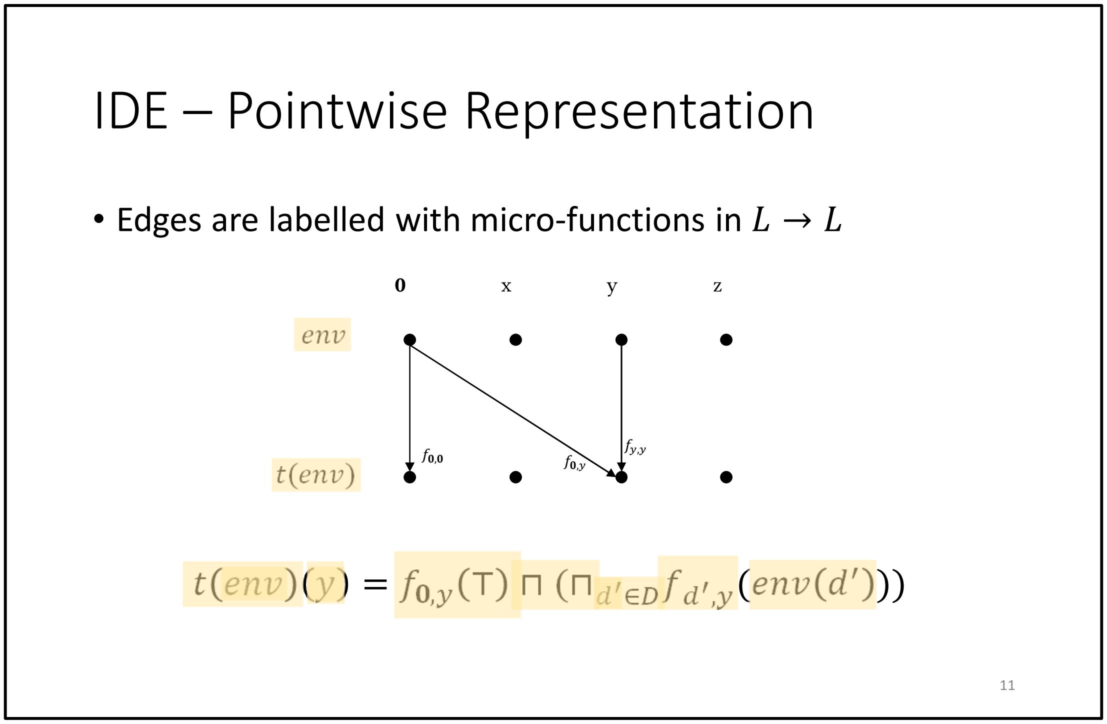
IDE – Formal Definition

$$P = \langle G^*, D, L, M_{Env} \rangle$$

Meet-over-valid-paths solution:

$$MVP_{IDE}(P) = \lambda n. \sqcap_{p \in VP(n)} M_{Env}(p)(\top_{Env})$$

- Formally, IDE is specified by a 4-tuple
    - G* is the supergraph
    - D is a finite set
    - L is the lattice
    - M_Env assigns environment transformers to each edge of the supergraph
- Solution is also a meet-over-valid-paths solution
    - Very similar to IFDS
    - Difference is using M_Env instead of M_F, and initializing with Top_Env instead of emptyset

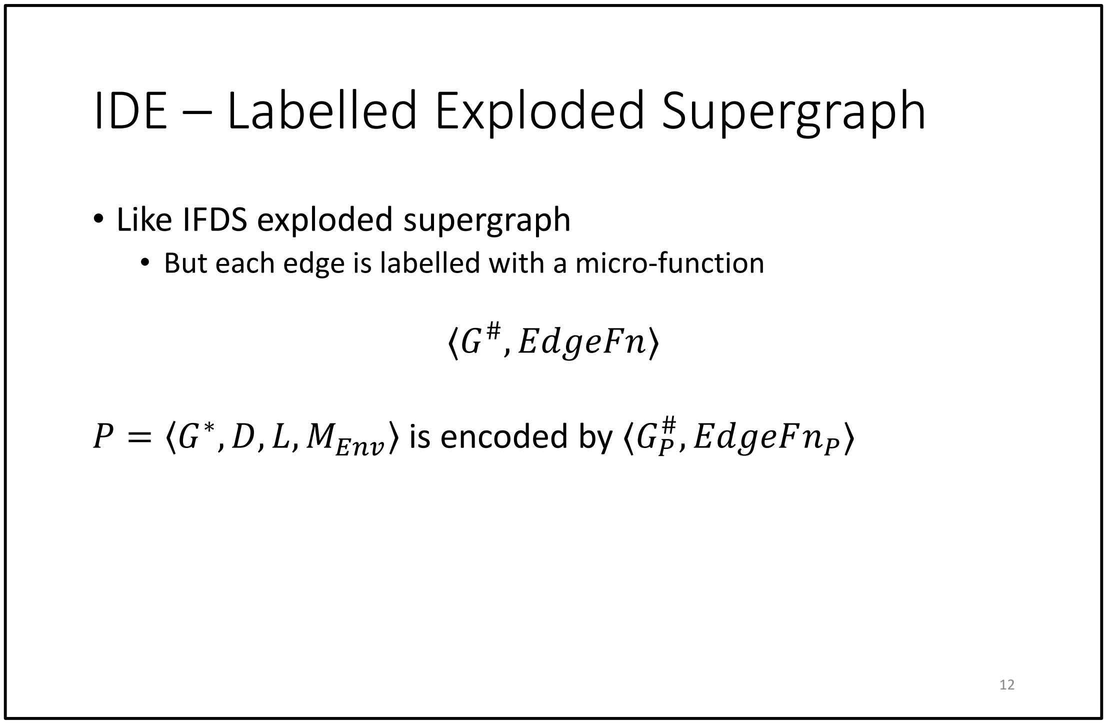


IDE – Pointwise Representation

- Edges are labelled with micro-functions in $L \to L$

$$t(env)(y) = f_{\mathbf{0},y}(\top) \sqcap (\sqcap_{d' \in D} f_{d',y}(env(d')))$$

- IDE has a *pointwise representation*, similar to the IFDS representation relation
  - Like the IFDS bipartite graph, but each edge is labelled with a micro-function
  - E.g., g_0, g_1, g_2
- Idea is that we can take the lattice value an element d is mapped to, and get the “output lattice value”
  - By doing this over all elements d and taking the meet, we can reconstruct the updated environment



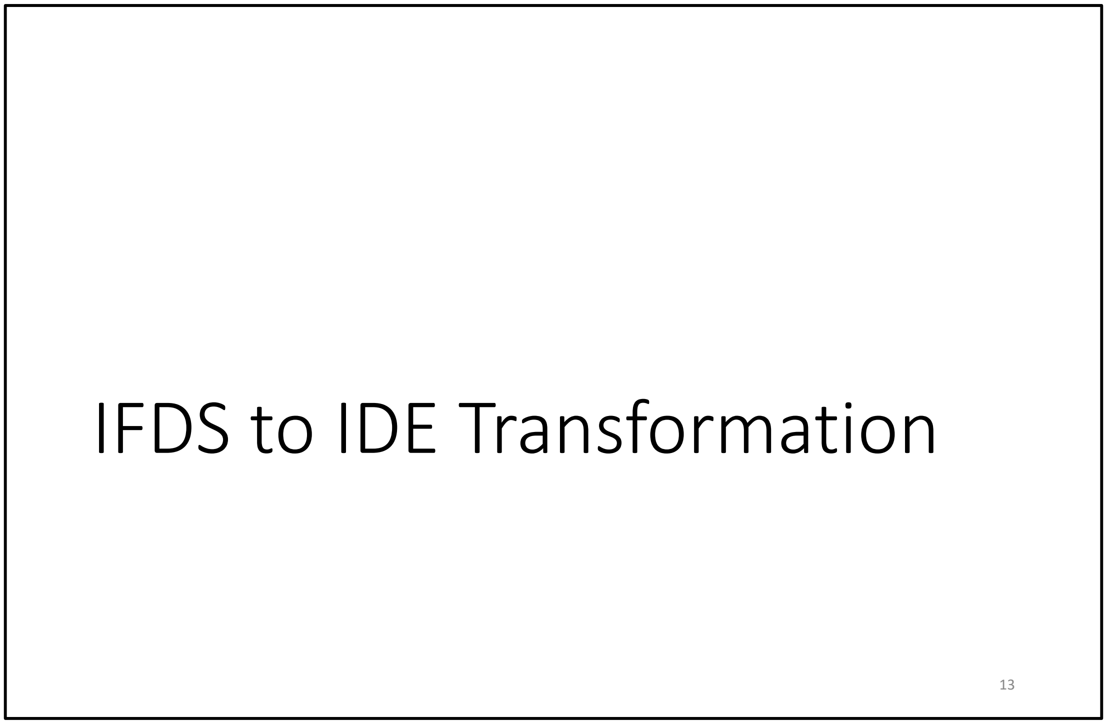
IDE – Labelled Exploded Supergraph

- Like IFDS exploded supergraph
  - But each edge is labelled with a micro-function

$$\langle G^{\#}, EdgeFn \rangle$$

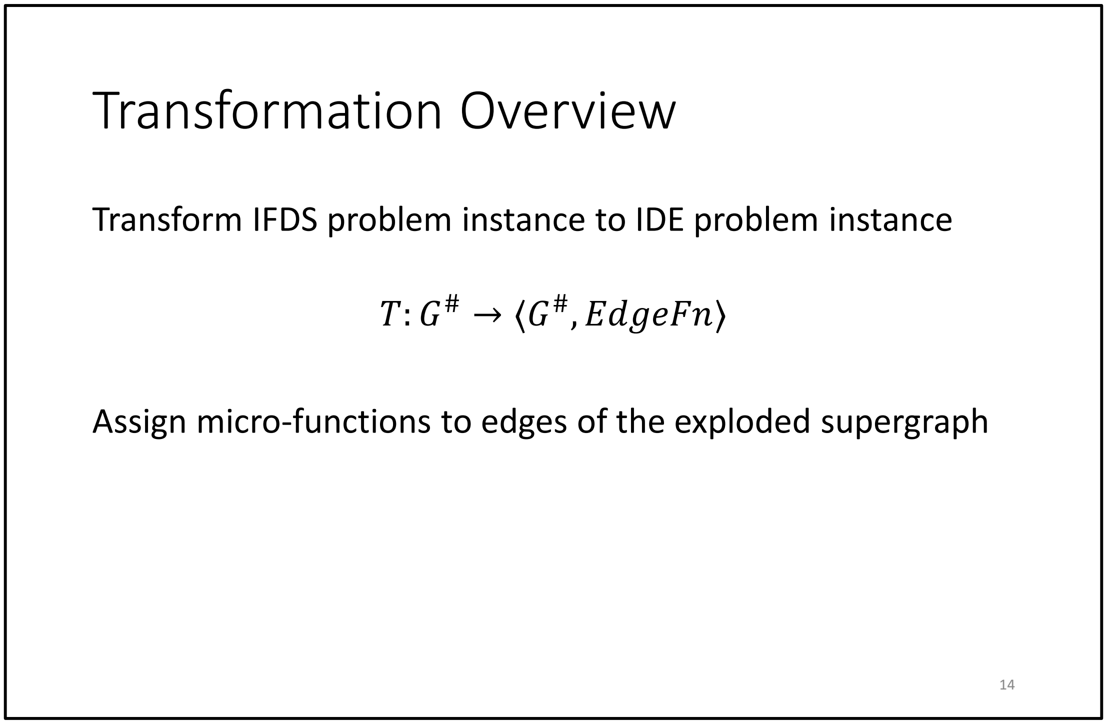
$P = \langle G^{*}, D, L, M_{Env} \rangle$ is encoded by $\langle G^{\#}_{P}, EdgeFn_{P} \rangle$

- Again, we can stitch the pointwise representations together
  - Form an exploded supergraph, where each edge is labelled by a micro-function
- This is a representation of an IDE problem instance
- To solve, require two phases
  - Run the graph reachability algorithm to determine which nodes are reachable
    - This also composes the micro-functions along the path
  - Then apply the composed micro-function to the dataflow fact
  - Assumes micro-function composition and application can be done in constant time



IFDS to IDE Transformation

20:00 (+8:00) mark



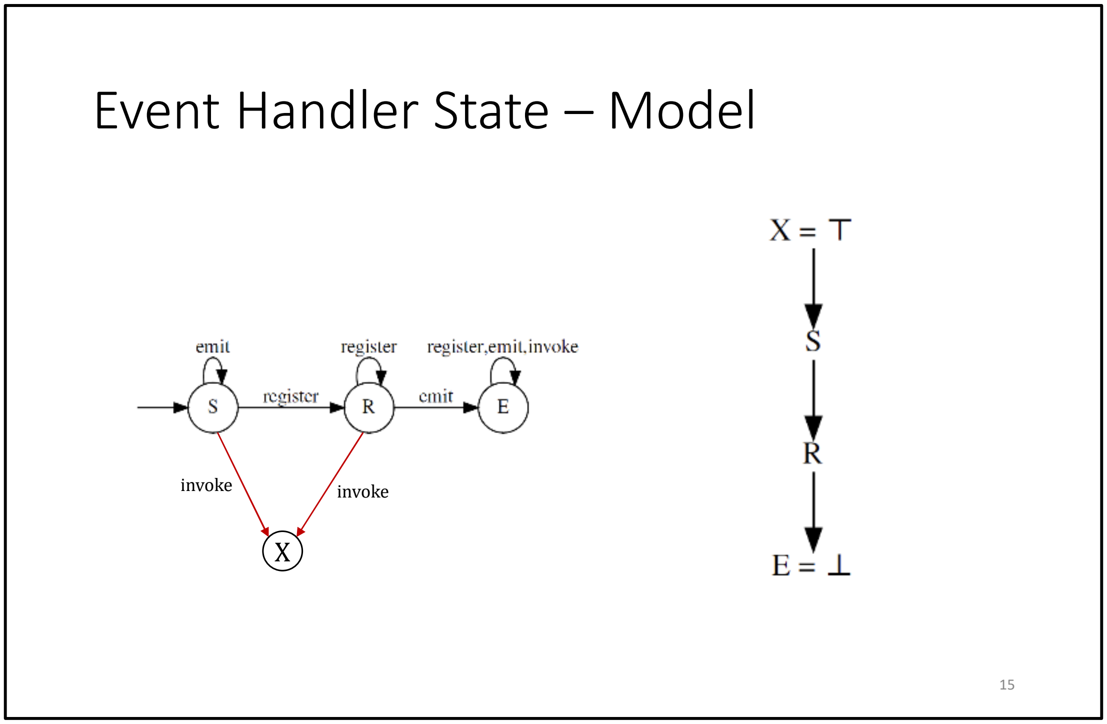
Transformation Overview

Transform IFDS problem instance to IDE problem instance

$$T: G^{\#} \rightarrow \langle G^{\#}, EdgeFn \rangle$$

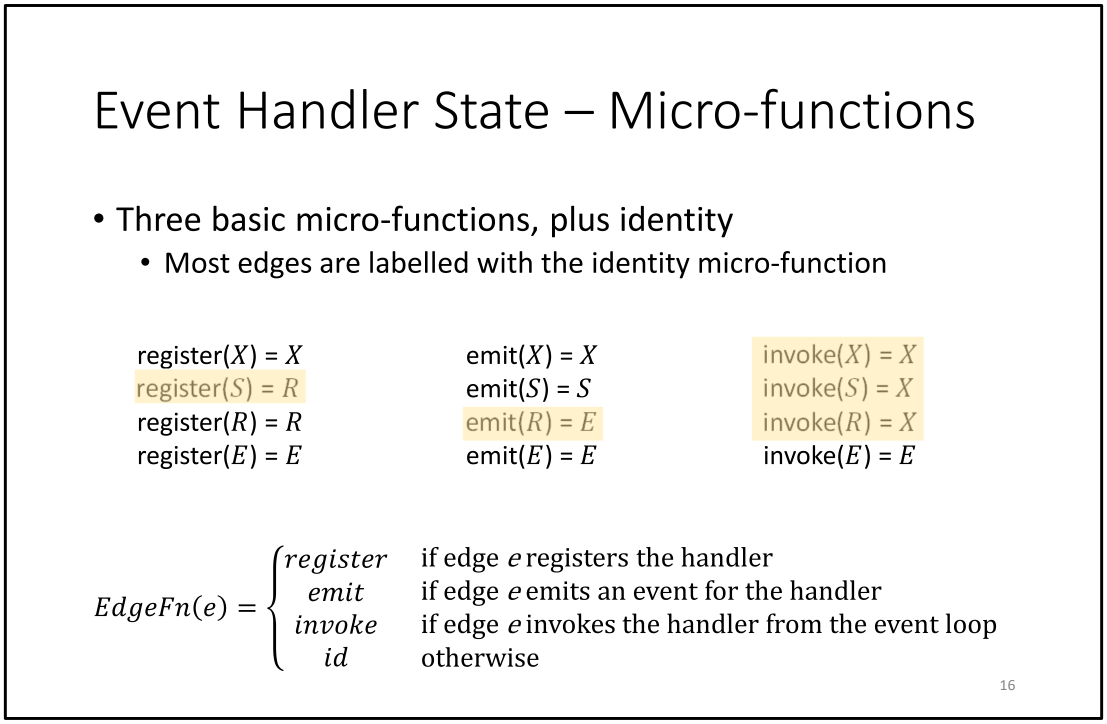
Assign micro-functions to edges of the exploded supergraph

- Our goal is to transform an IFDS problem to an IDE problem
  - We are given some existing IFDS analysis
  - The transformation works on the exploded supergraph and adds labels
  - Does not change the exploded supergraph (= program being analyzed + original analysis)
- Idea is to use the micro-functions to encode the event handler operations (register, emit, invoke)
- IFDS analysis asks “is dataflow fact d present at node n?”
- IDE analysis asks “what lattice value is associated with element d at node n?”
  - In this case, what is the state of the event handler?
  - If it is “infeasible” we can ignore the result on this path



Event Handler State – Model

- For now, assume a single event handler in the program
- An event handler has three states: start (S), registered (R), and emitted (E)
  - Transition actions are “register”, “emit”, and “invoke”
- Note that the handler can get “stuck” if it invokes from S or R
  - This never happens in a real program execution
  - But we model it with the infeasible (X) state for an analysis
- Lattice ordering is for merging results from two branches
  - E.g. one branch has “infeasible” and the other branch has “registered”
  - We have to be conservative and assume the state after the branch is “registered”



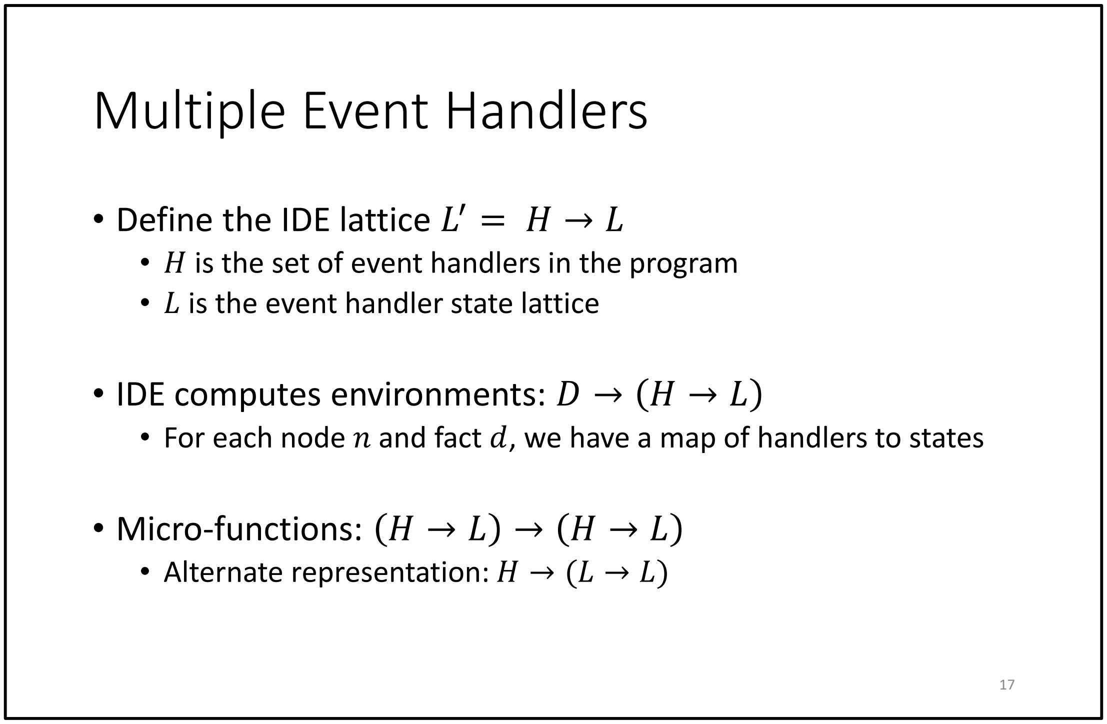
Event Handler State – Micro-functions

- Three basic micro-functions, plus identity
  - Most edges are labelled with the identity micro-function

register(X) = X	emit(X) = X	invoke(X) = X
register(S) = R	emit(S) = S	invoke(S) = X
register(R) = R	emit(R) = E	invoke(R) = X
register(E) = E	emit(E) = E	invoke(E) = E

$$EdgeFn(e) = \begin{cases} register & \text{if edge } e \text{ registers the handler} \\ emit & \text{if edge } e \text{ emits an event for the handler} \\ invoke & \text{if edge } e \text{ invokes the handler from the event loop} \\ id & \text{otherwise} \end{cases}$$

- Our analysis requires three basic micro-functions, plus the identity
  - These micro-functions correspond to the event handler operations: register, emit, and invoke
  - Most edges are labelled with the identity micro-function
- Register: only update S state to R
- Emit: only update R state to E
- Invoke: update non-E states to X
- EdgeFn: micro-functions correspond to edges that involve an event handler operation
- As IDE algorithm traverses the exploded supergraph, it composes these micro-functions along the paths
  - Initial state is S
  - E.g. invoke(emit(register(S))) = E is OK, but invoke(register(S)) = X is not
- Notice that we can represent each micro-function as a 4-tuple
  - Only 8 bits needed to represent 256 functions

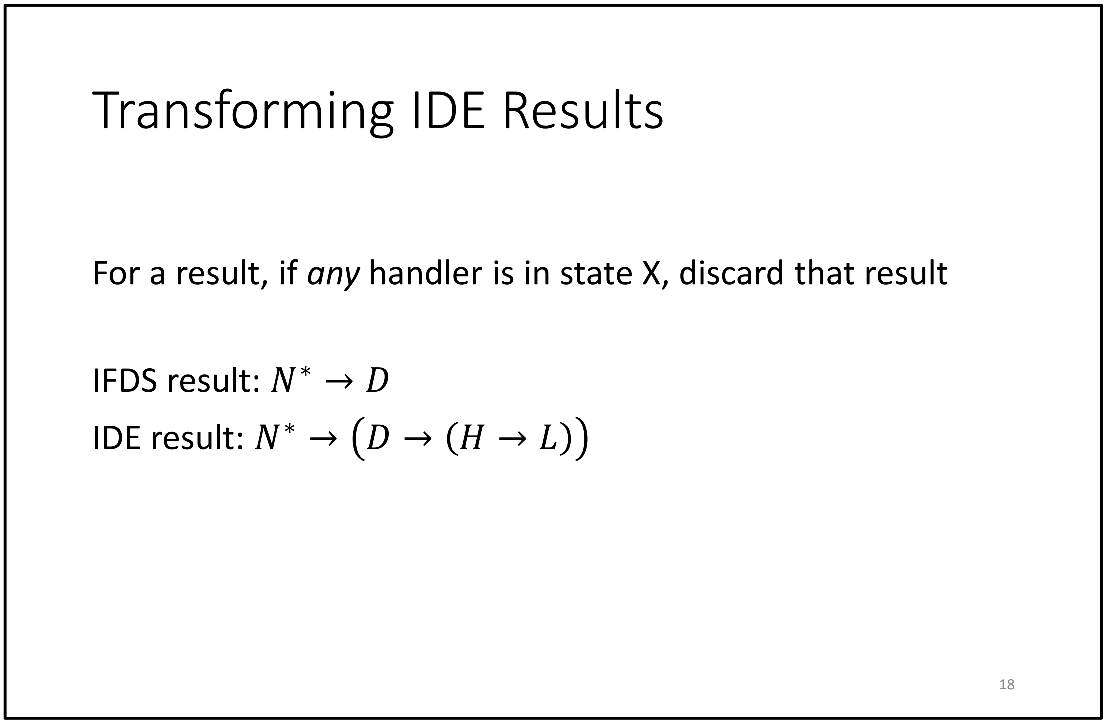


Multiple Event Handlers

- Define the IDE lattice $L' = H \to L$
  - H is the set of event handlers in the program
  - L is the event handler state lattice
- IDE computes environments: $D \to (H \to L)$
  - For each node n and fact d, we have a map of handlers to states
- Micro-functions: $(H \to L) \to (H \to L)$
  - Alternate representation: $H \to (L \to L)$

17

- Now we need to support multiple event handlers
- We use the lattice L’ = H \to L
  - H is the set of event handlers in the program
  - L is the event handler state lattice we just saw
- Recall: IDE computes an environment D \to L’ at each node n
  - Now, for each node n, we have an environment D \to (H \to L)
  - For a given dataflow fact at node n, we have a map m : H \to L with states for each event handler
- So micro-functions are in (H \to L) \to (H \to L) which are hard to represent
  - But note that the state of an event handler does not depend on any other handler
  - So we can represent micro-functions in H \to (L \to L)
  - We have a map from event handlers to micro-functions in L \to L
- So when EdgeFn assigns micro-functions, it has to pick the micro-function for the correct event handler

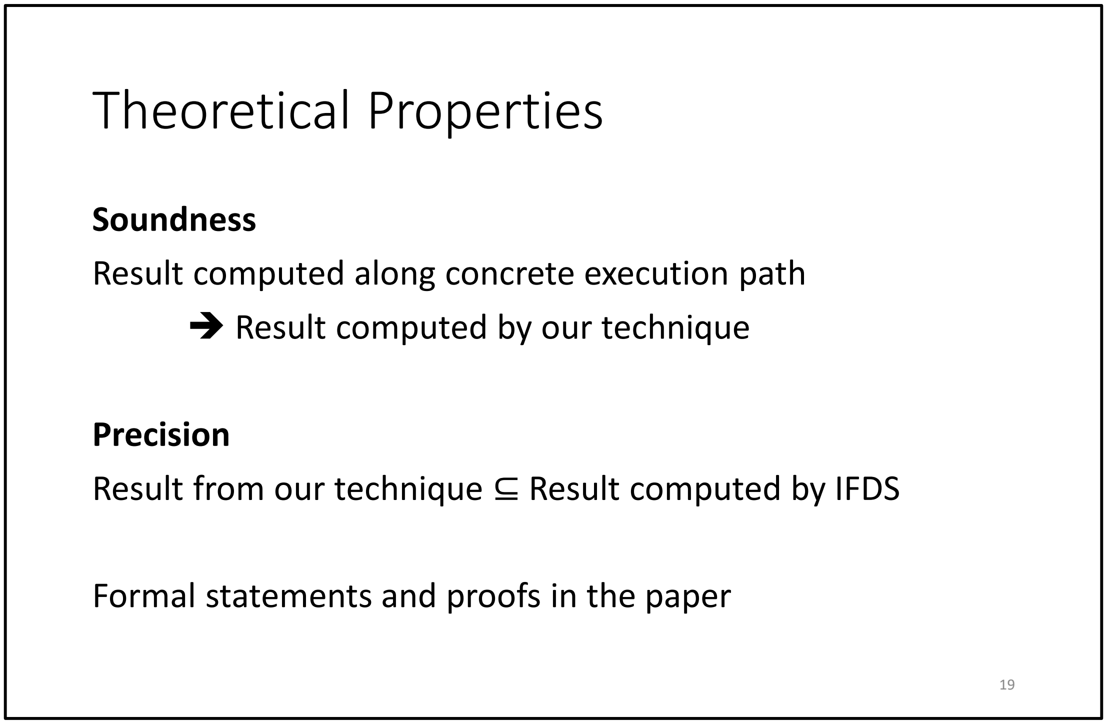


Transforming IDE Results

For a result, if *any* handler is in state X, discard that result

IFDS result: $N^* \to D$
IDE result: $N^* \to \big(D \to (H \to L)\big)$

- Idea of transformation: each result d has a map of event handlers to states
    - If any event handler has state X, then the result is impossible
    - It was computed along some path that did not respect the ordering constraints of an event handler
    - So discard that result
- Formally, we have an “untransformation”
    - IFDS result is a map from nodes to elements in D
    - IDE result is a map from nodes to environments, where an environment maps elements of D to event handler maps
- Untransformation returns a map from nodes to elements in D
    - For the given node, look it up in the IDE result, and find all elements d where all handlers are not in state X



Theoretical Properties

Soundness

Result computed along concrete execution path

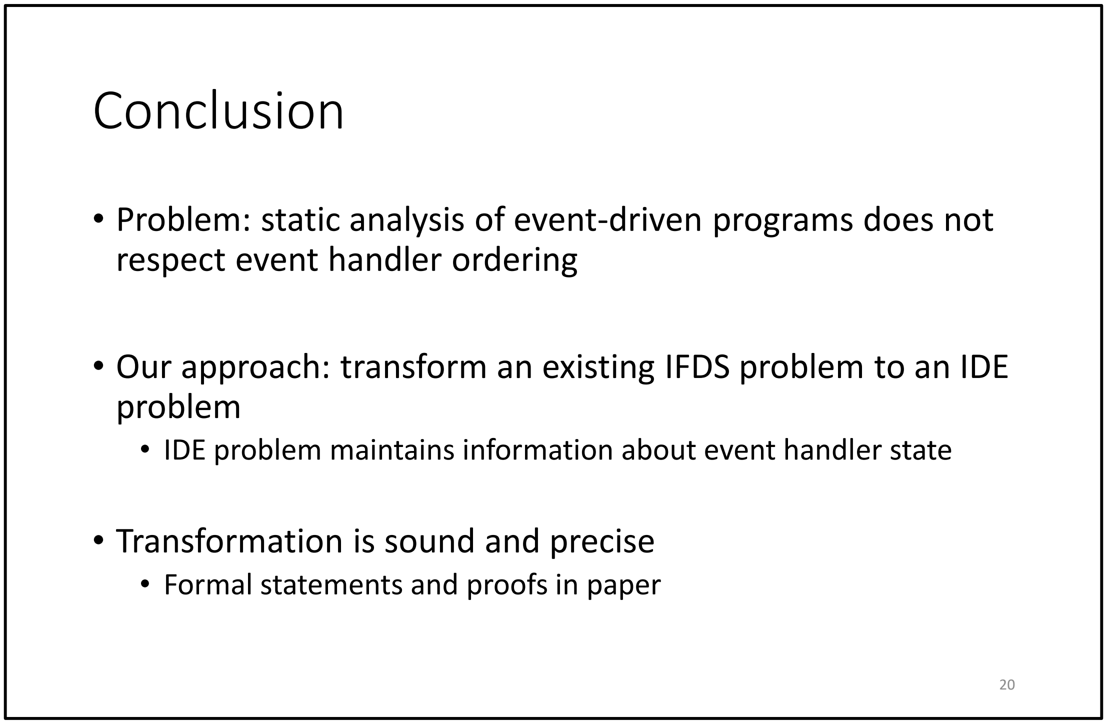
➔ Result computed by our technique

Precision

Result from our technique ⊆ Result computed by IFDS

Formal statements and proofs in the paper

- Transformation is sound
  - Consider running the IFDS analysis on a *concrete* path.
  - Consider our technique: transform IFDS to IDE, run it, untransform it.
  - Results on the concrete path will be returned by our technique.
- Transformation is precise
  - Consider running the IFDS analysis on an entire program.
  - Consider our technique: transform IFDS to IDE, run it, untransform it.
  - Our results will be a (non-strict) subset of the IFDS results.

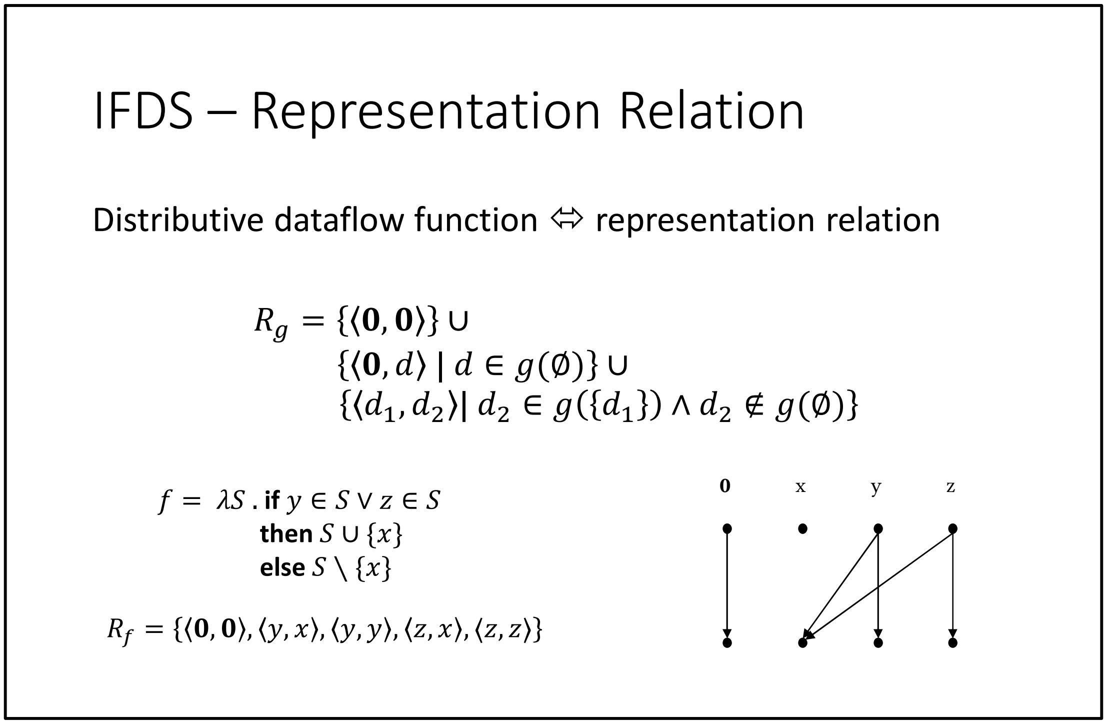


Conclusion

- Problem: static analysis of event-driven programs does not respect event handler ordering
- Our approach: transform an existing IFDS problem to an IDE problem
  - IDE problem maintains information about event handler state
- Transformation is sound and precise
  - Formal statements and proofs in paper

30:00 (+10:00) mark

Extra Slides



IFDS – Representation Relation

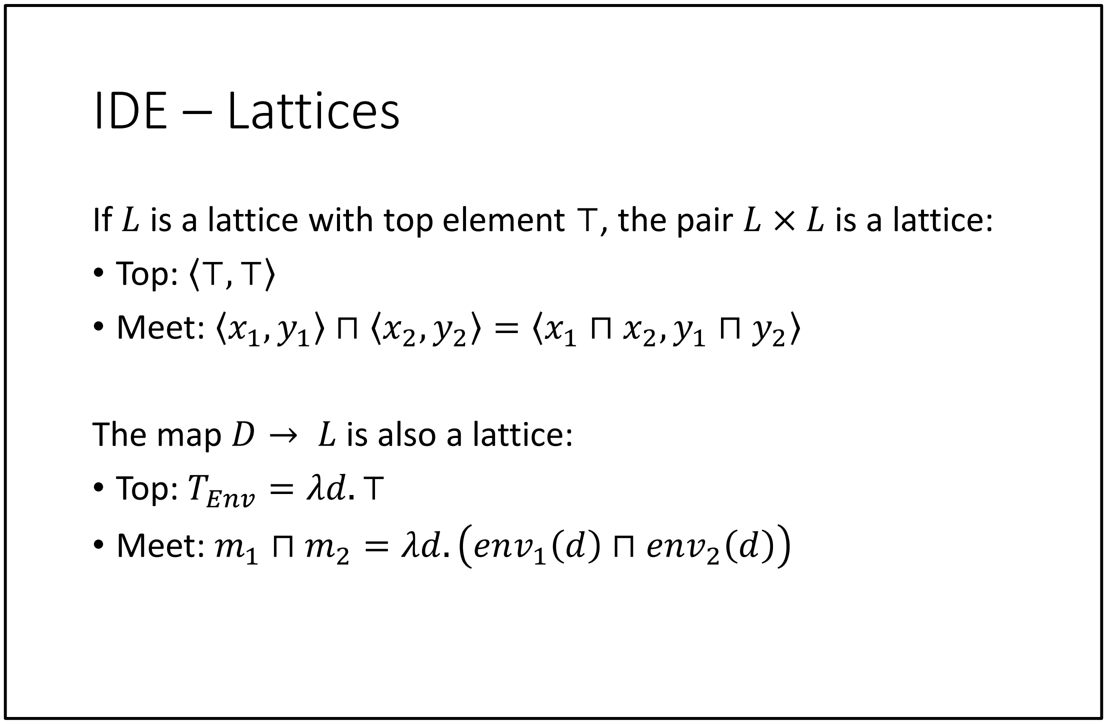
Distributive dataflow function ⇔ representation relation

$$\begin{aligned} R_g = {} & \{\langle \mathbf{0},\mathbf{0}\rangle\} \cup \\ & \{\langle \mathbf{0}, d\rangle \mid d \in g(\emptyset)\} \cup \\ & \{\langle d_1, d_2\rangle \mid d_2 \in g(\{d_1\}) \land d_2 \notin g(\emptyset)\} \end{aligned}$$

```
$f = \lambda S$ . **if** $y \in S \lor z \in S$
    **then** $S \cup \{x\}$
    **else** $S \setminus \{x\}$
```

$$R_f = \{\langle \mathbf{0},\mathbf{0}\rangle, \langle y,x\rangle, \langle y,y\rangle, \langle z,x\rangle, \langle z,z\rangle\}$$

- This is the formal definition for the representation relation
- <0,0> is always in the relation and roughly corresponds to the empty set
- <0,d> corresponds to facts “created” or in the “gen set”
- <d_1, d_2> corresponds to inputs-outputs, but there is a “subsumption” to avoid excess edges
    - If the output is already “created” by the empty set, then we don’t care what its input is
    - This condition does not apply for the current example



IDE – Lattices

If L is a lattice with top element $\top$, the pair $L \times L$ is a lattice:

- Top: $\langle \top, \top \rangle$
- Meet: $\langle x_1, y_1 \rangle \sqcap \langle x_2, y_2 \rangle = \langle x_1 \sqcap x_2, y_1 \sqcap y_2 \rangle$

The map $D \rightarrow L$ is also a lattice:

- Top: $T_{Env} = \lambda d. \top$
- Meet: $m_1 \sqcap m_2 = \lambda d. \left(env_1(d) \sqcap env_2(d) \right)$

- Given a lattice L, we can build up “bigger” lattices, like pairs and maps
- For a pair, the top value is when both elements of the pair are top
    - The meet is done pointwise, for each element
    - We can extend this to n-tuples
- For a map D to L, it doesn’t matter what D is
    - The top value is the map that maps every element to top
    - The meet of two maps is done pointwise
    - Note that we could interpret the map as a function
- General idea is to do the operations per-element

Transforming IDE Results

For a result, if *any* handler is in state X, discard that result

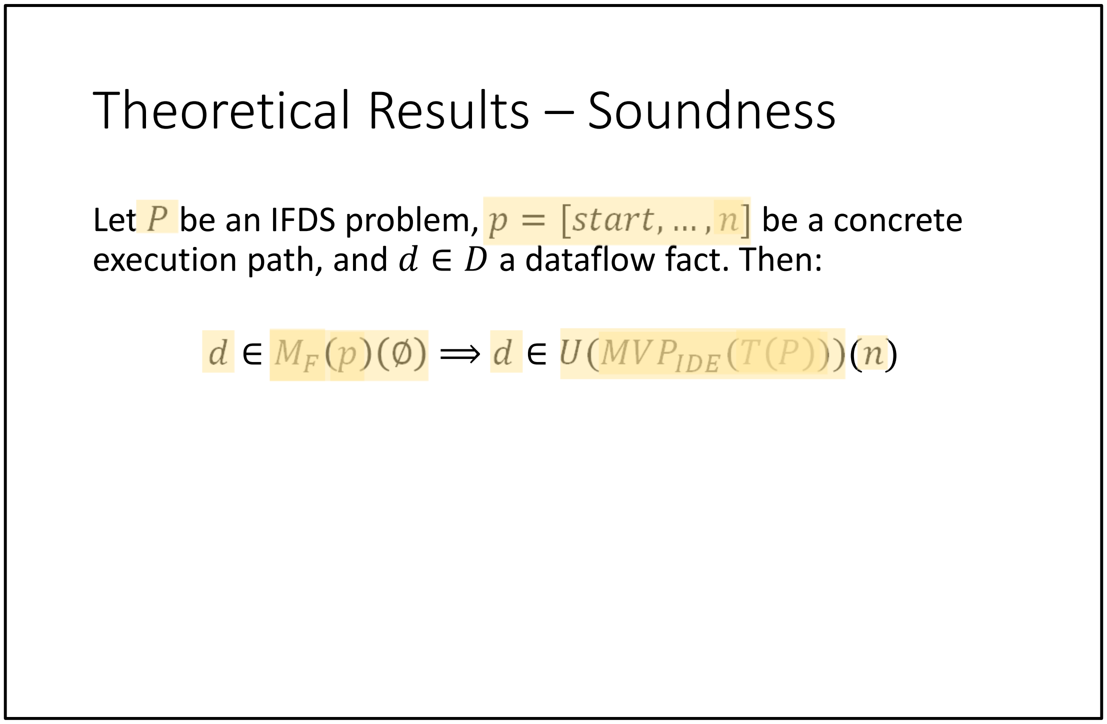
IFDS result: $N^* \to D$

IDE result: $N^* \to \big(D \to (H \to L)\big)$

"Untransform" function U applied to IDE result R:

$$U(R) = \lambda n. \{d \mid \forall h \in H \,.\, R(n)(d)(h) \neq X\}$$

- Idea of transformation: each result d has a map of event handlers to states
    - If any event handler has state X, then the result is impossible
    - It was computed along some path that did not respect the ordering constraints of an event handler
    - So discard that result
- Formally, we have an "untransformation"
    - IFDS result is a map from nodes to elements in D
    - IDE result is a map from nodes to environments, where an environment maps elements of D to event handler maps
- Untransformation returns a map from nodes to elements in D
    - For the given node, look it up in the IDE result, and find all elements d where all handlers are not in state X

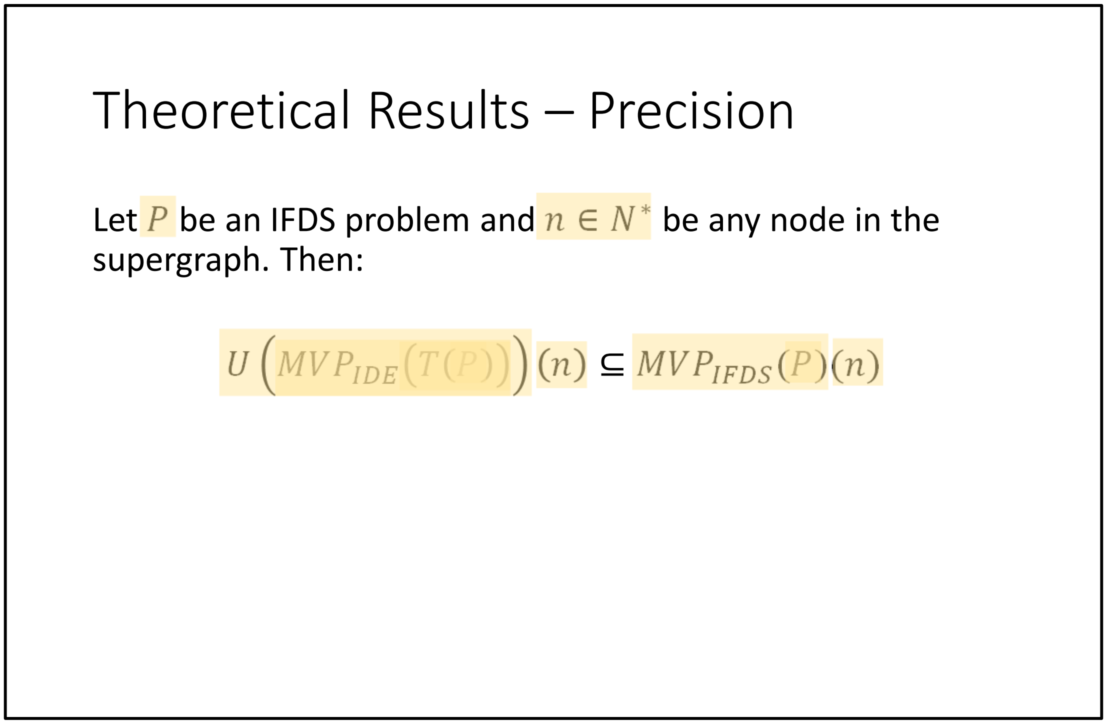


Theoretical Results – Soundness

Let P be an IFDS problem, $p = [start, \dots, n]$ be a concrete execution path, and $d \in D$ a dataflow fact. Then:

$$d \in M_F(p)(\emptyset) \Longrightarrow d \in U(MVP_{IDE}(T(P)))(n)$$

- The transformation is sound
- Consider an IFDS problem P and a concrete execution path p
    - Take the dataflow functions along p, compose them, and apply it to the empty set
        - Will get some result set containing dataflow facts d
    - These results will be returned by our transformation
        - I.e. transform P to IDE, solve it, untransform it, and then get its result



Theoretical Results – Precision

Let P be an IFDS problem and $n \in N^*$ be any node in the supergraph. Then:

$$U\left(MVP_{IDE}\left(T(P)\right)\right)(n) \subseteq MVP_{IFDS}(P)(n)$$

- The transformation is precise
- Take an IFDS problem P, and a node n
  - On the RHS, we solve the IFDS problem and get some set of results
  - On the LHS, we transform to IDE, solve that IDE problem to get results, and untransform those results
  - Then we compare the two results for the given node n
  - The LHS will be a subset of the RHS, because we filtered out the infeasible results