Hierarchical models of provenance

Peter Buneman
James Cheney
Egor Kostylev

University of Edinburgh
TaPP, June 15, 2012

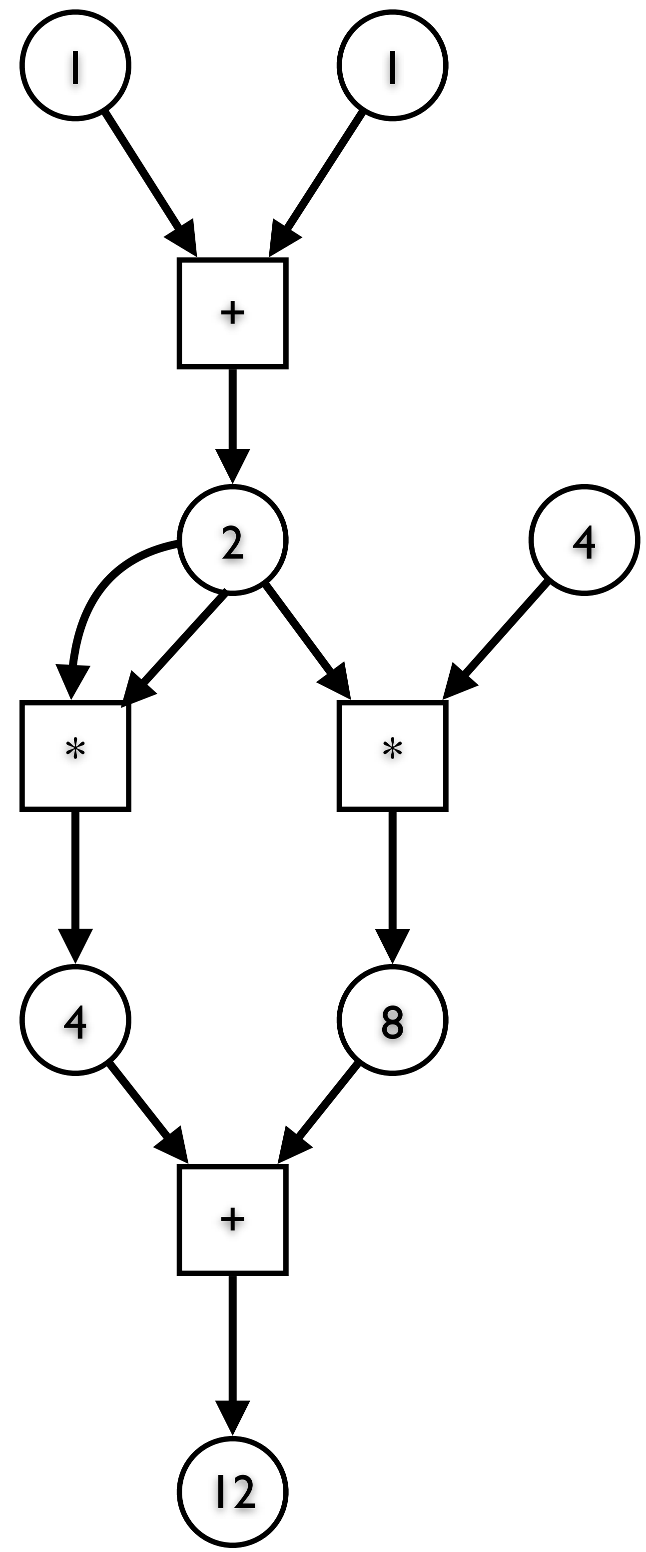
Motivation

- Many provenance systems track "flat" provenance
- Some track provenance at multiple granularity levels (in different ways)
  - e.g. ZOOM, Kepler, probably others
- Our goals:
  - Formal, high-level model of "hierarchical" provenance
  - Understand interplay between control/data abstractions and provenance models

OPM lite

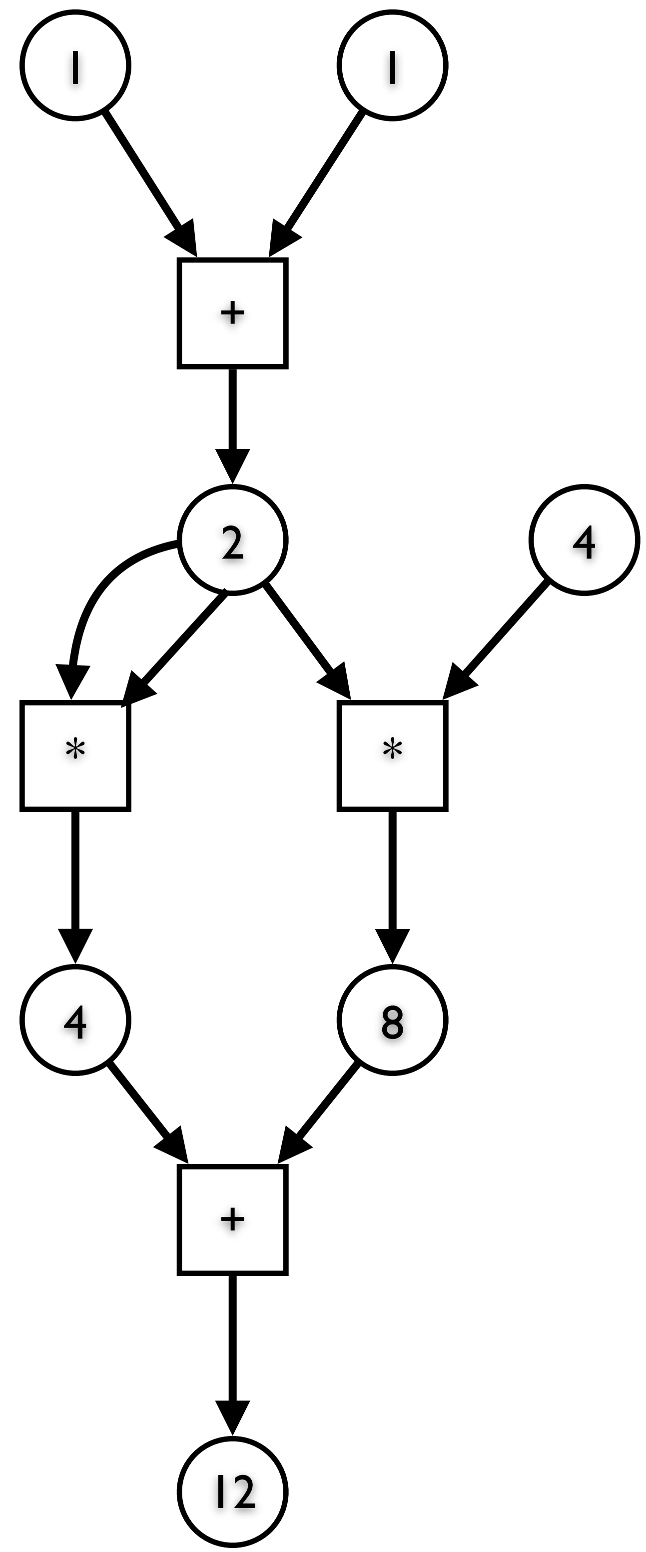
- Simplified OPM: bipartite DAGs
  - Process nodes P
  - Artifact nodes A
  - Edges $E \subseteq (P \times A) \cup (A \times P)$
  - Labels on process, artifacts and edges

```
def f(x)   = x+1
    g(x,y) = h(x) + x*y
    h(z)   = z*z
in g(f(1), 4)
```



```
def f(x)    = x+1
    g(x,y) = h(x) + x*y
    h(z)    = z*z
in g(f(1), 4)
```

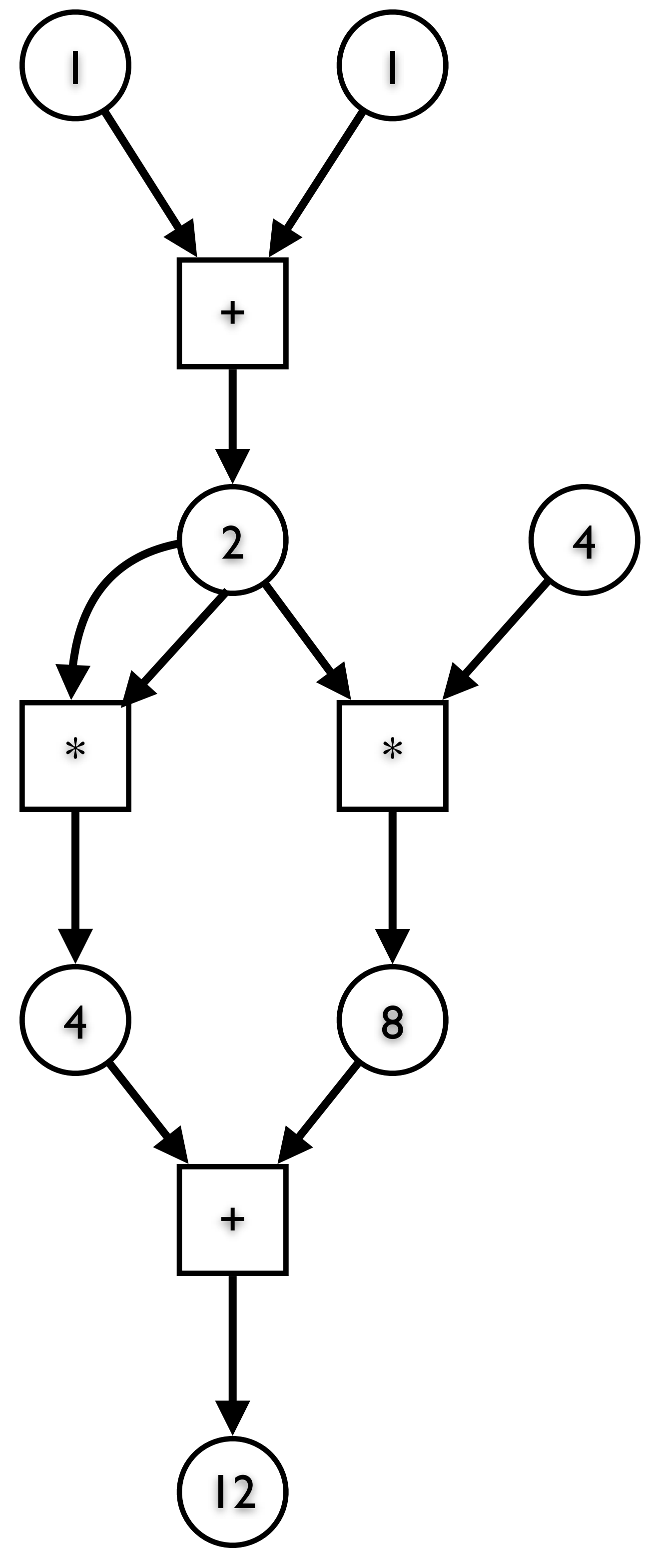
Note: This **discards** a lot of information!



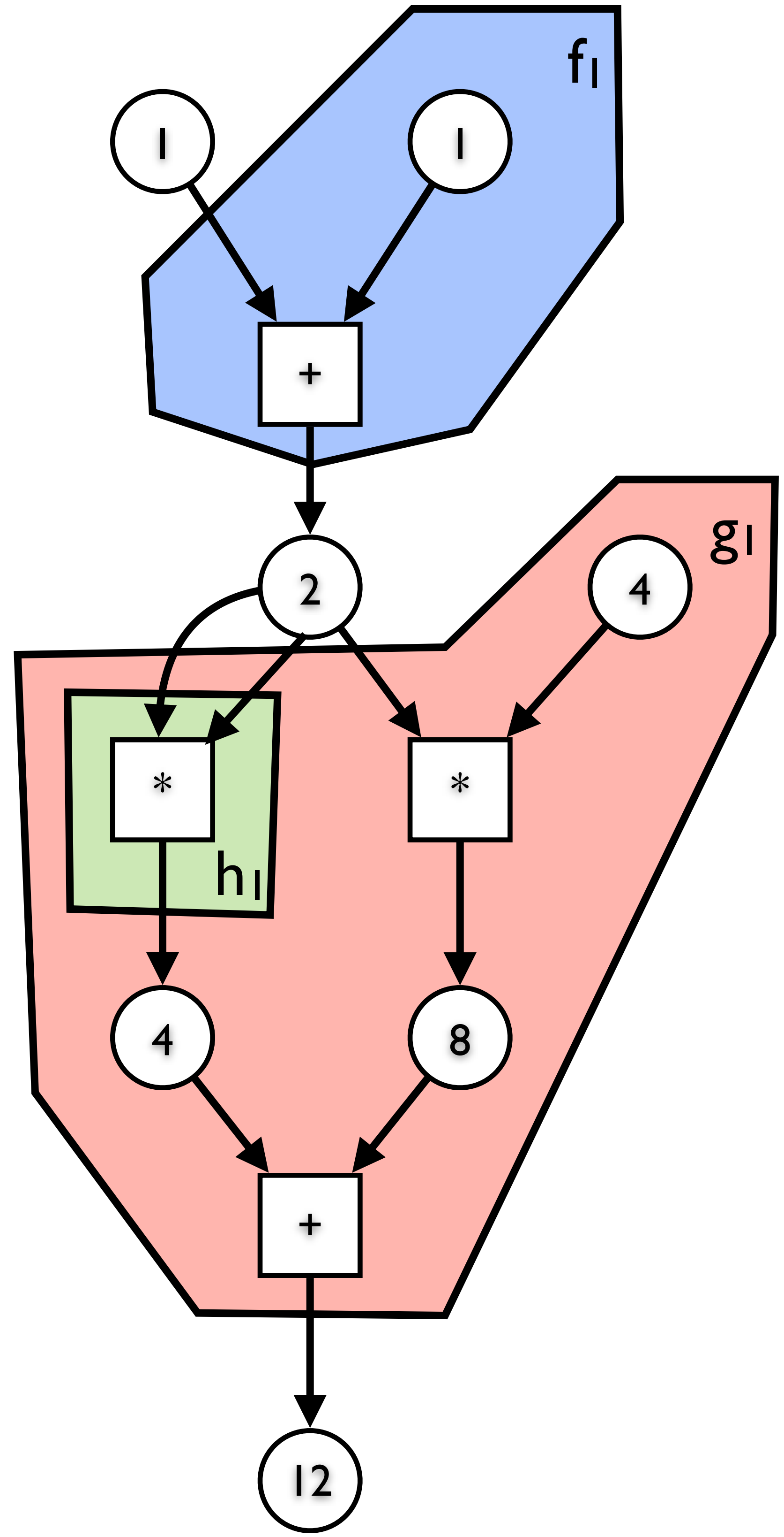
"Hierarchical" OPM

- Augment OPM graph structure with *call tree*
- Tree T = (V,F) with nodes labeled by "higher-level" processes
- Mapping $\Omega : V \rightarrow (P \cup A)$
  - Requires if $(f,g) \in F$ then $\Omega(f) \supseteq \Omega(g)$
    - note reversal!
    - $\Omega(\text{root}) = (P \cup A)$
  - Further requirements: $\Omega(f)$ is a contiguous, "sub-OPM" graph.
    - Formalized more carefully in paper.

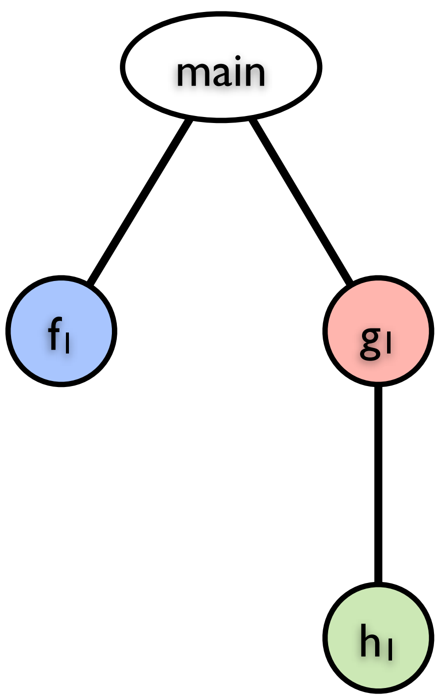
```
def f(x)    = x+1
    g(x,y) = h(x) + x*y
    h(z)    = z*z
in g(f(1), 4)
```

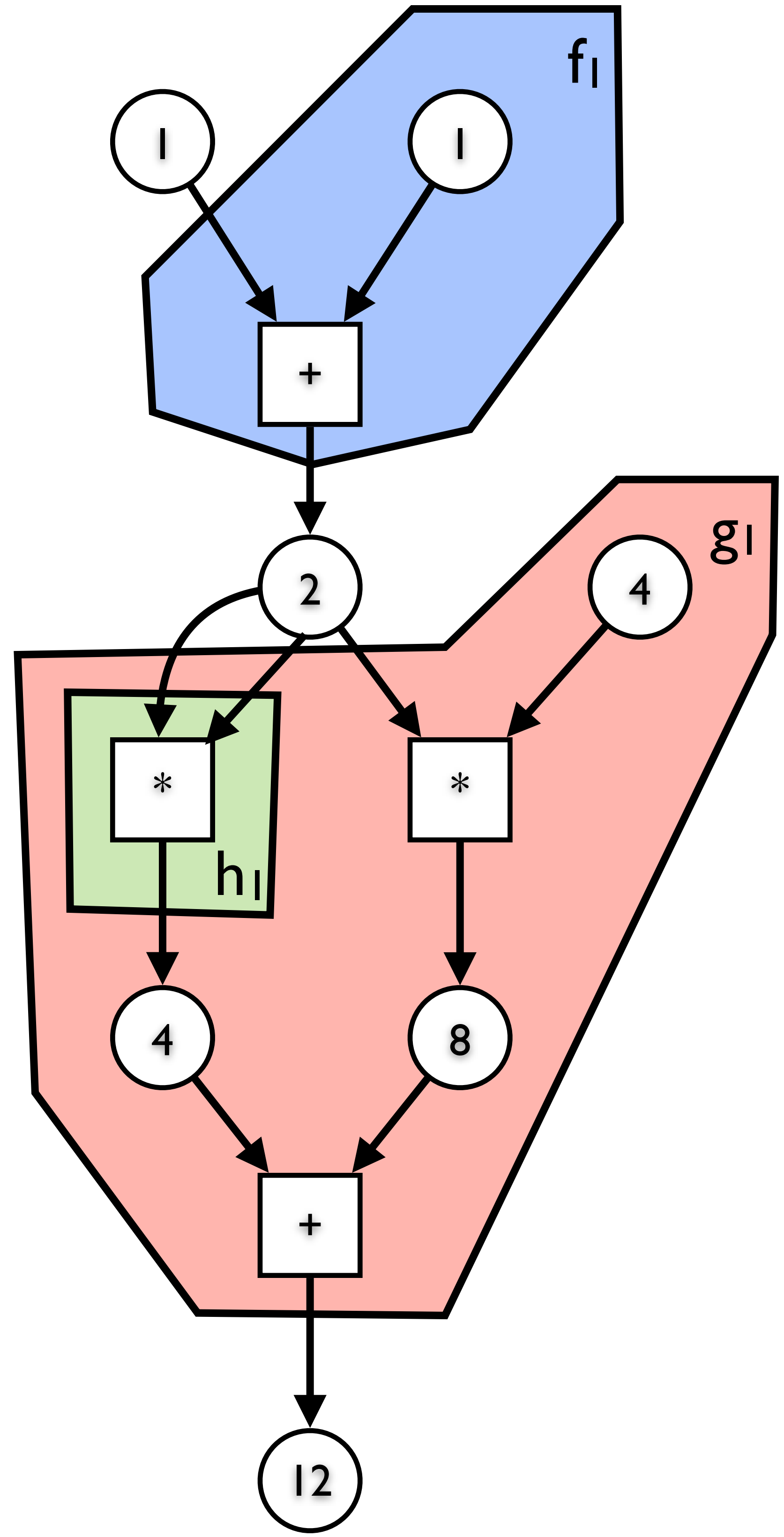


```
def f(x)    = x+1
    g(x,y) = h(x) + x*y
    h(z)    = z*z
in g(f(1), 4)
```



```
def f(x)    = x+1
    g(x,y) = h(x) + x*y
    h(z)    = z*z
in g(f(1), 4)
```

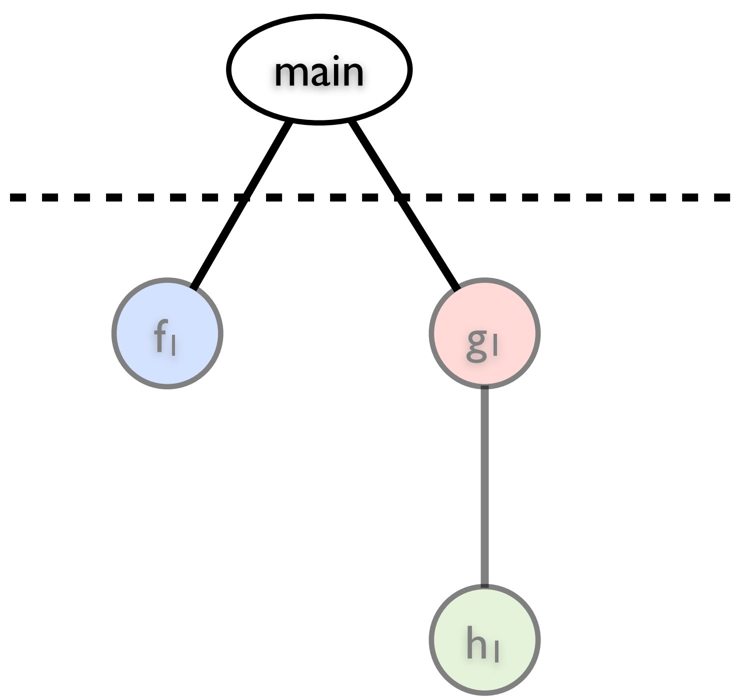


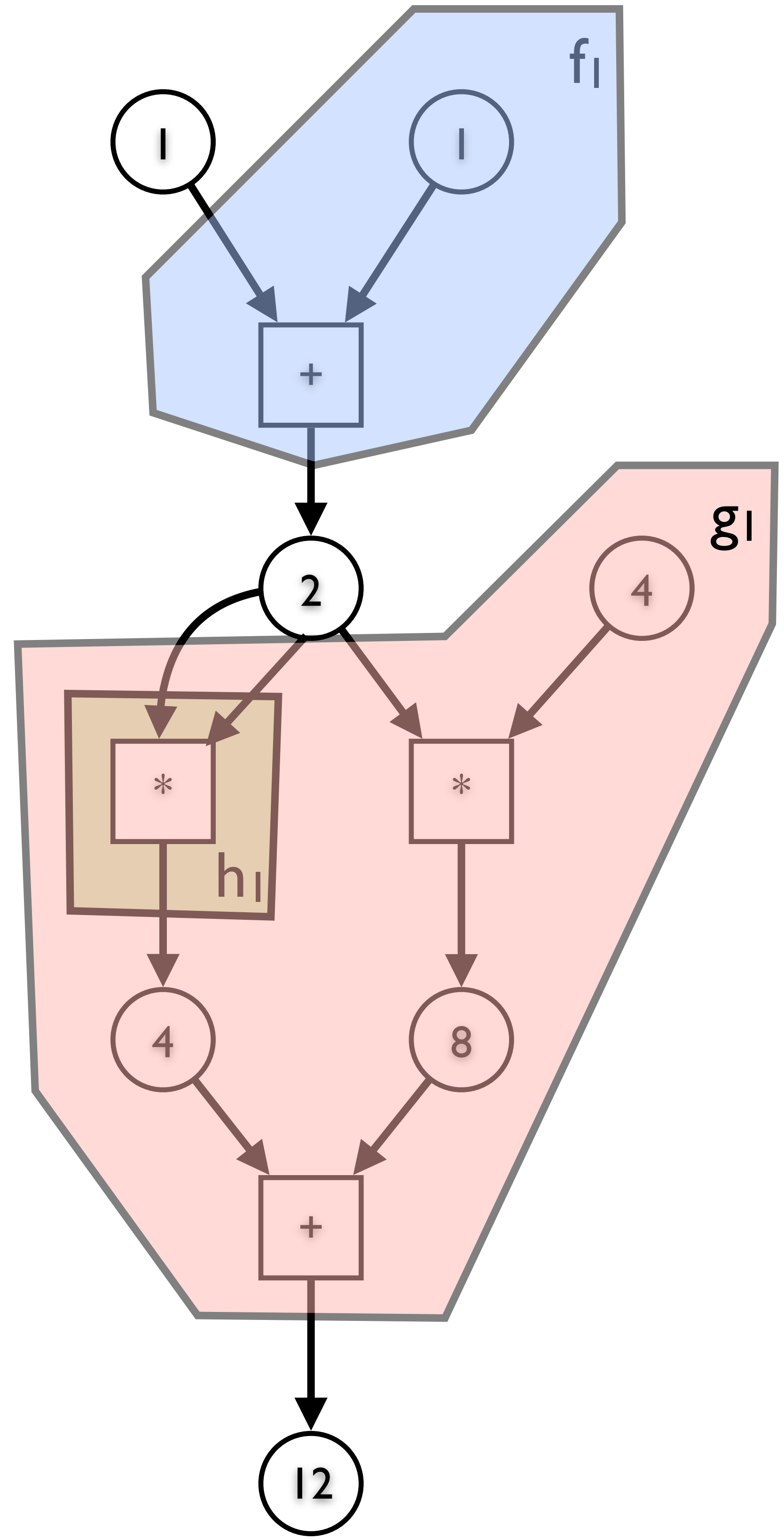


Views

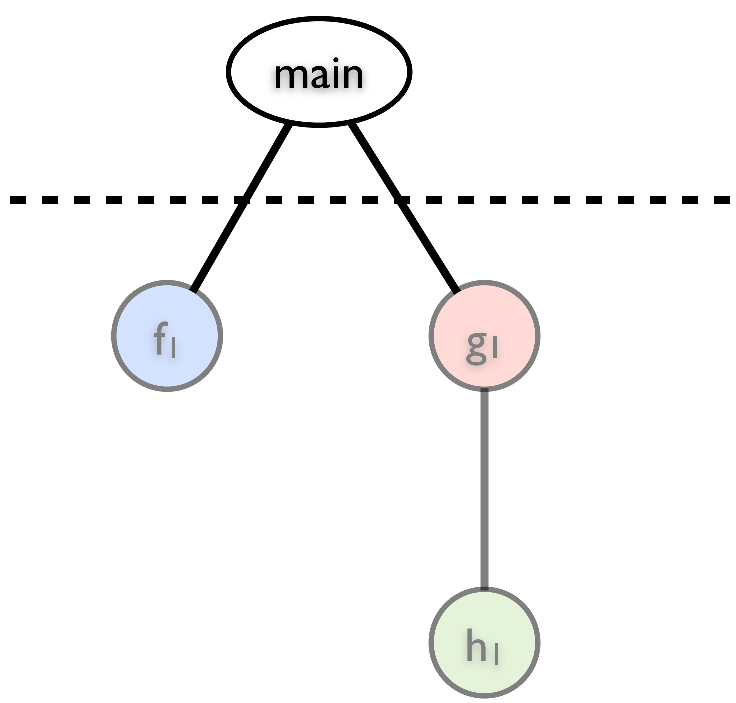
- Given a prefix-closed subtree S of T
- This induces a *view* of H
  - that is, an normal OPM graph
- obtained by "collapsing" the graph structure of calls not in S
- This makes sense (only) because of restrictions on call mapping Ω.
  - (details in paper - there are a few subtleties)

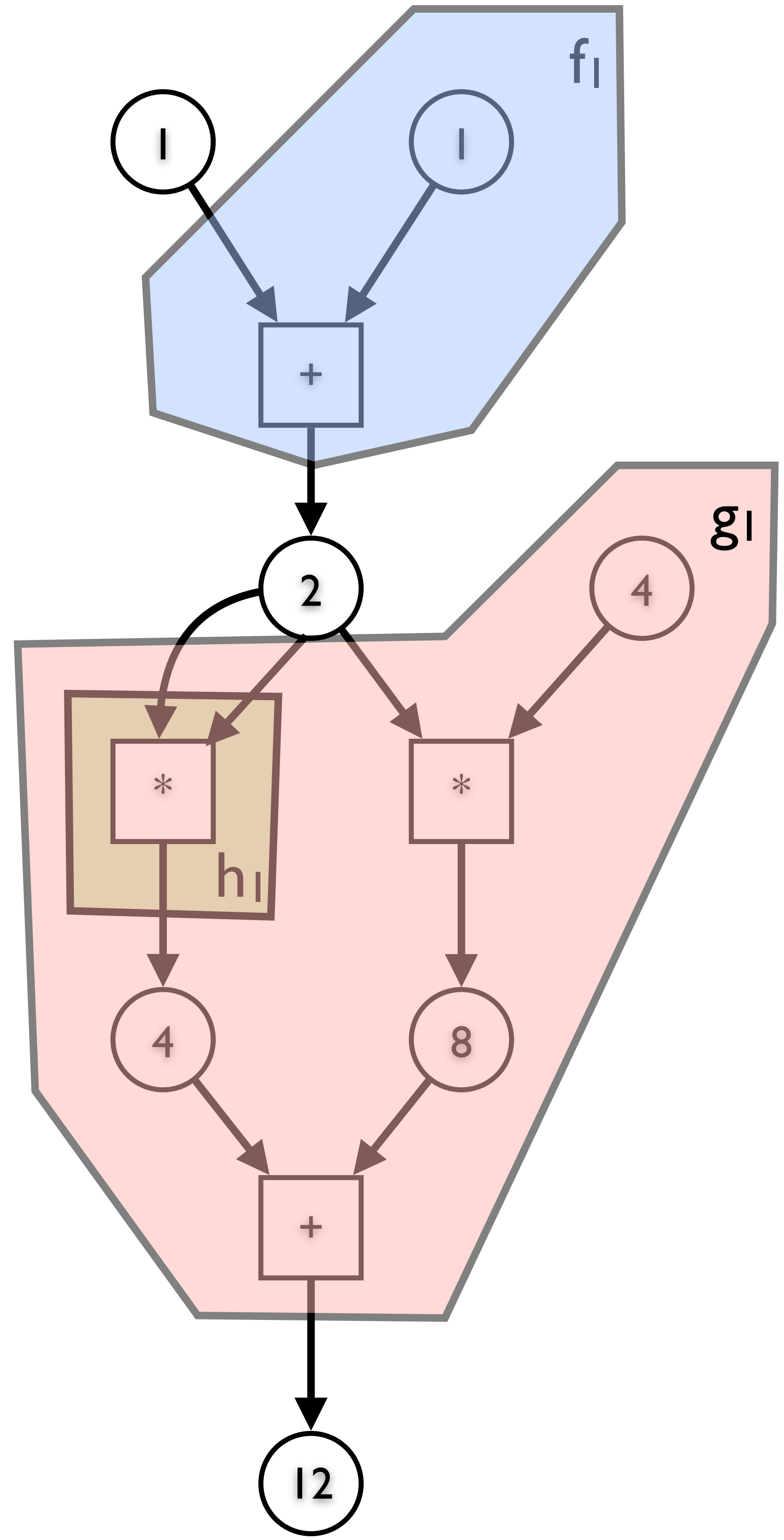
```
def f(x)    = x+1
    g(x,y) = h(x) + x*y
    h(z)   = z*z
in g(f(1), 4)
```

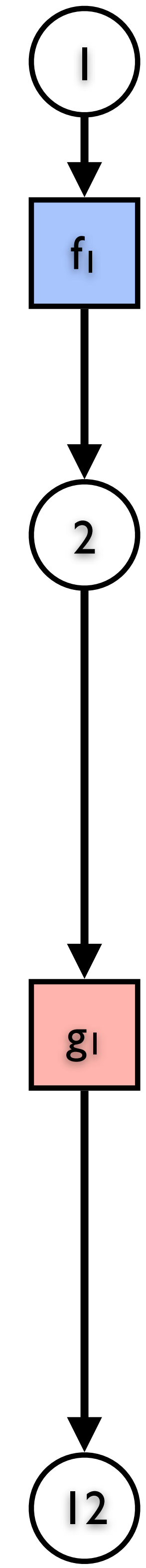




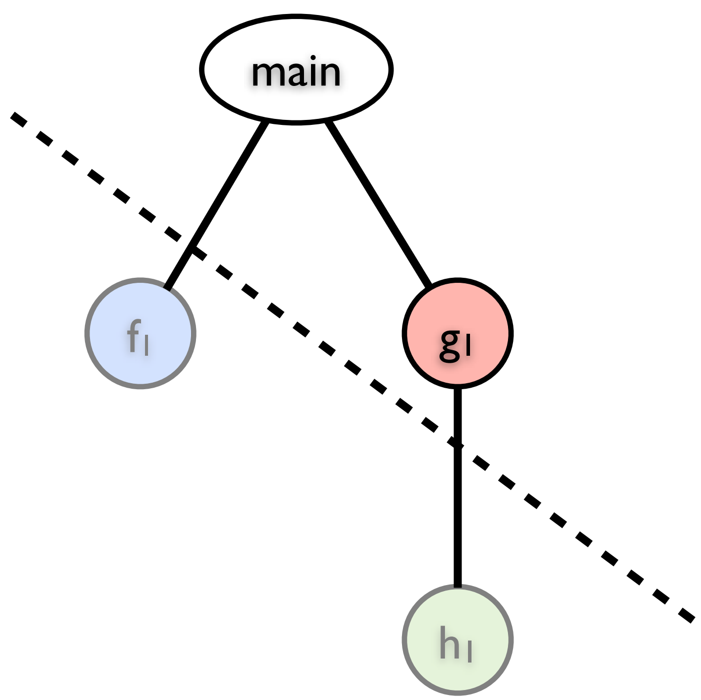
```
def f(x)    = x+1
    g(x,y) = h(x) + x*y
    h(z)    = z*z
in g(f(1), 4)
```

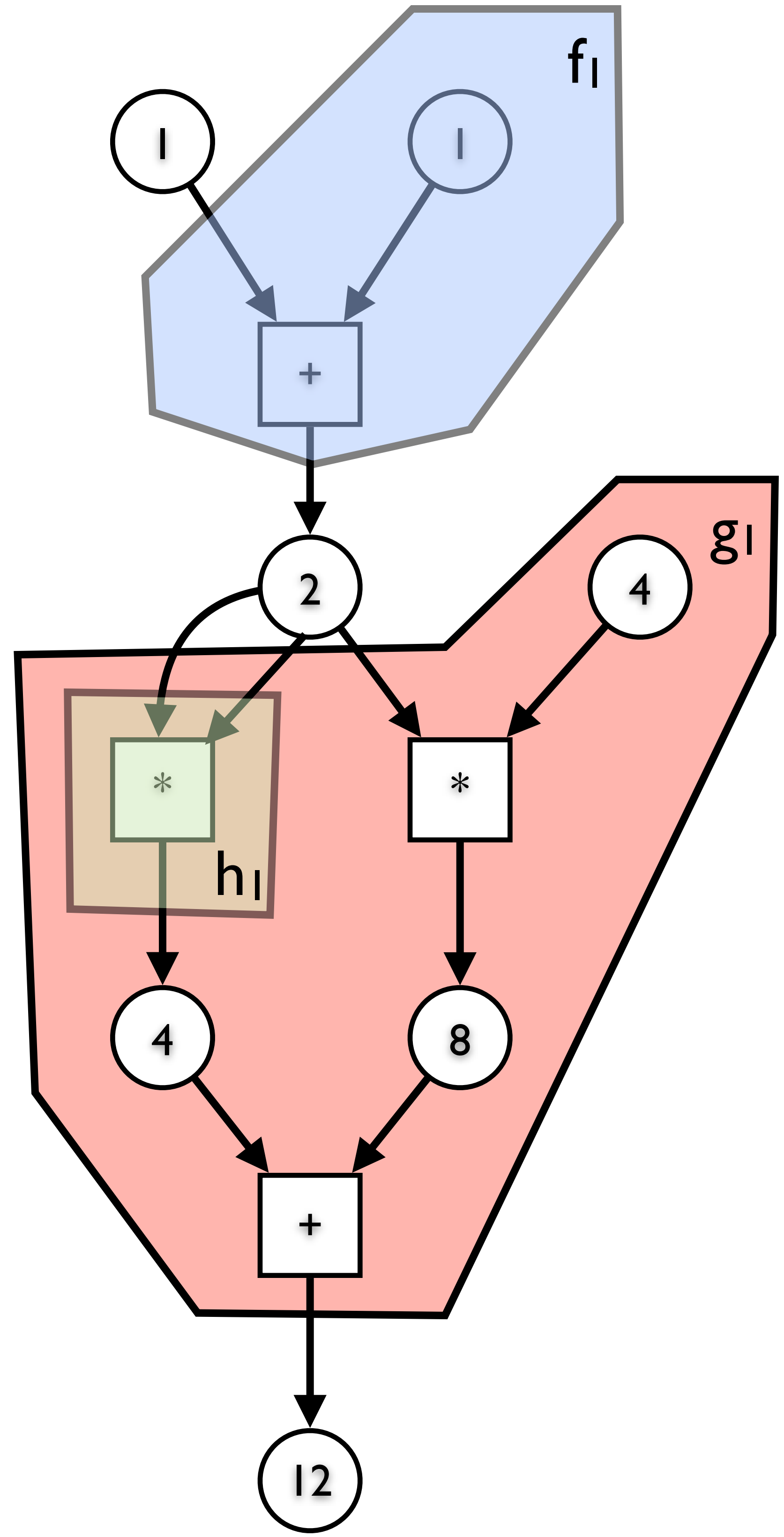


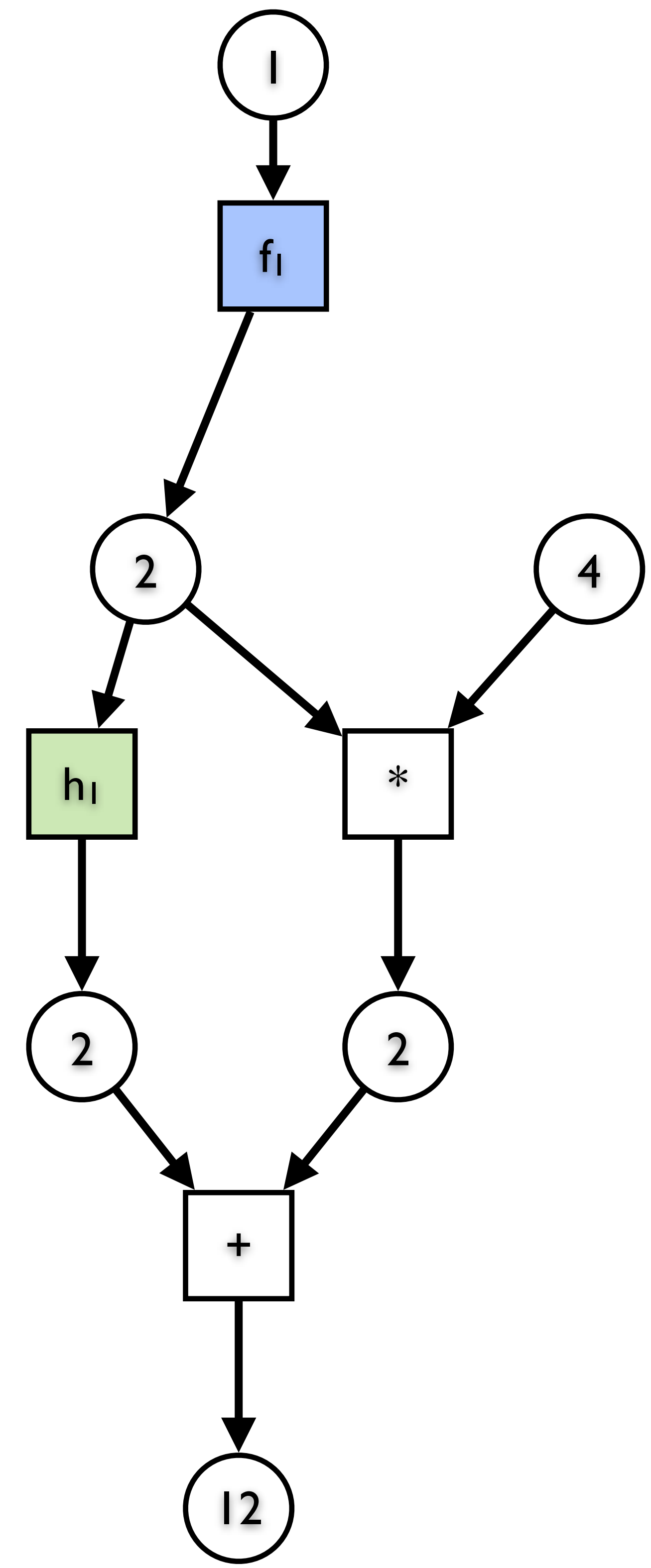




```
def f(x)   = x+1
    g(x,y) = h(x) + x*y
    h(z)   = z*z
in g(f(1), 4)
```







ProvL: Simple "workflows" (ie functional programs)

- We first consider workflows with function calls but little else...

$$e ::= c \mid x \mid \odot(\vec{e}) \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$$

- $\odot$ denotes an arbitrary primitive function
  - think +, *, -, etc.
- Let-binding allows for sharing
  - hence, expression basically isomorphic to graph

Adding function calls

- We allow (closed, first- order) function definitions, with calls:

$$e ::= \cdots \mid f(\vec{e})$$

$$\texttt{def}\ f_1(\vec{x}_1) = e_1, \ldots, f_m(\vec{x}_m) = e_m\ \texttt{in}\ e'$$

- Functions can be defined mutually recursively
- Function calls generate call tree nodes (mapped to appropriate subgraphs).

Adding lists, map

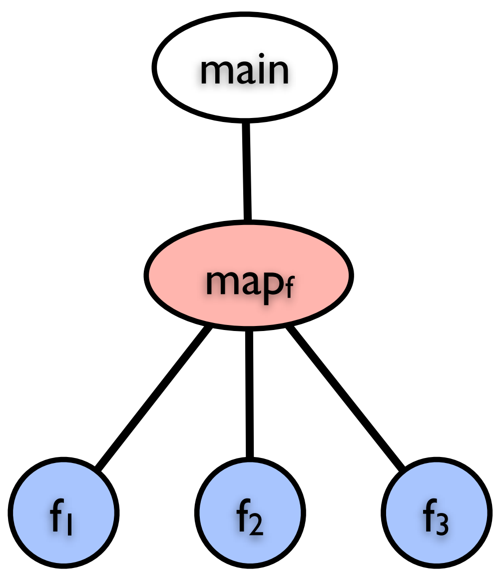
- Finally we consider lists and mapping

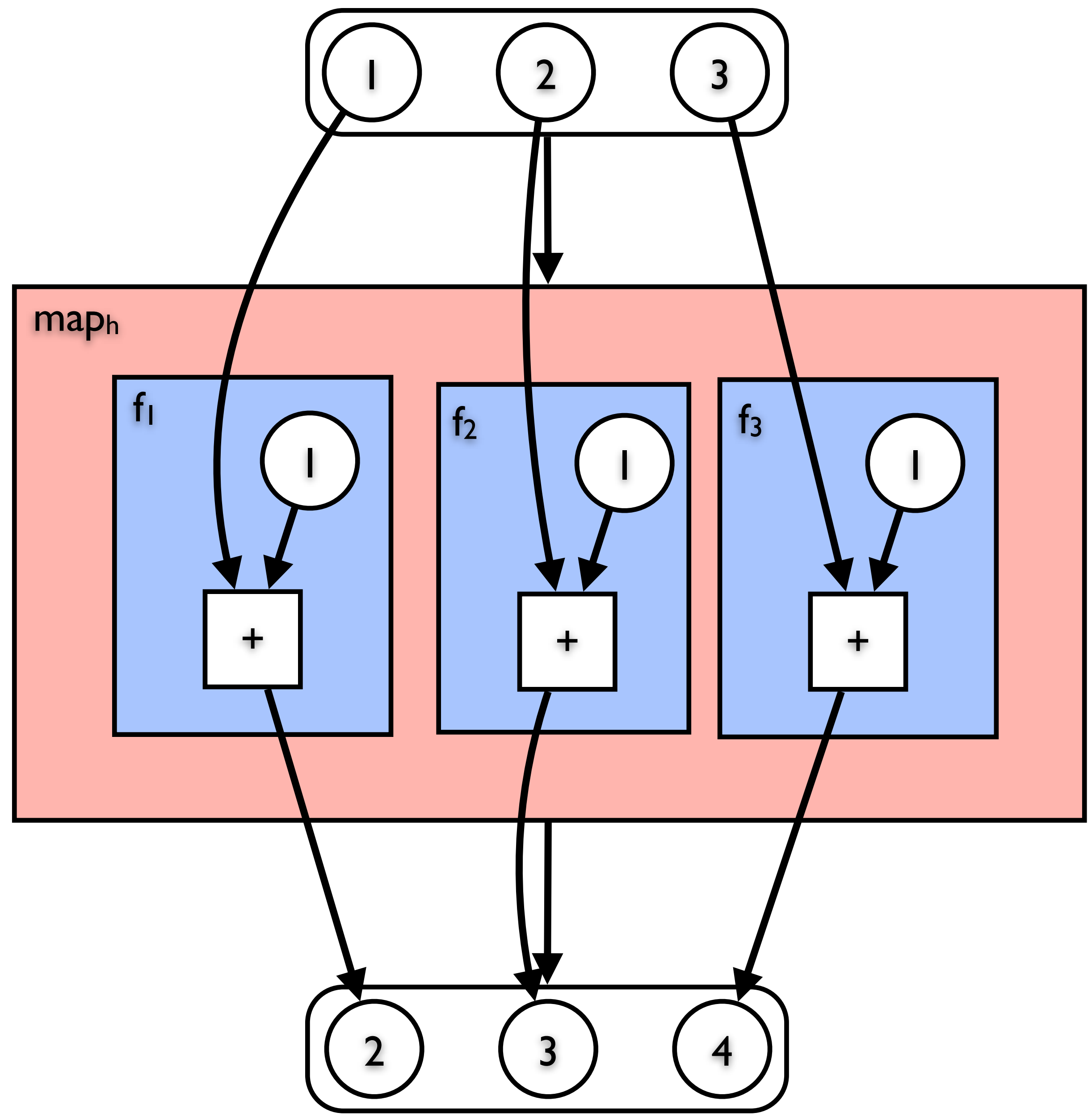
$$e ::= \;\cdots \mid \mathtt{map}_f(e)$$

- and allow `nil`, `cons` (and maybe others) as built-in functions
- Note that map is second-order - i.e. $\mathtt{map}_\mathtt{f}$ is a function for any `f`
- Map nodes link lists to lists, sub-call nodes map elements to elements

Map-incr example

```
def f(x)    = x + 1
in mapf([1,2,3])
```





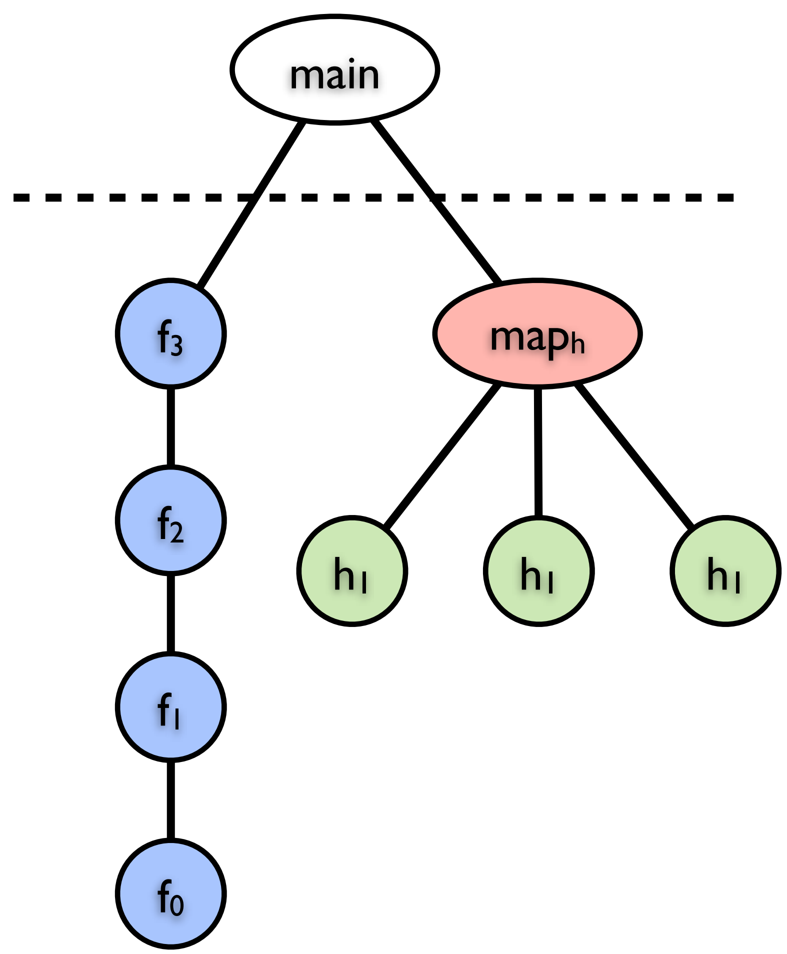
Adding conditionals

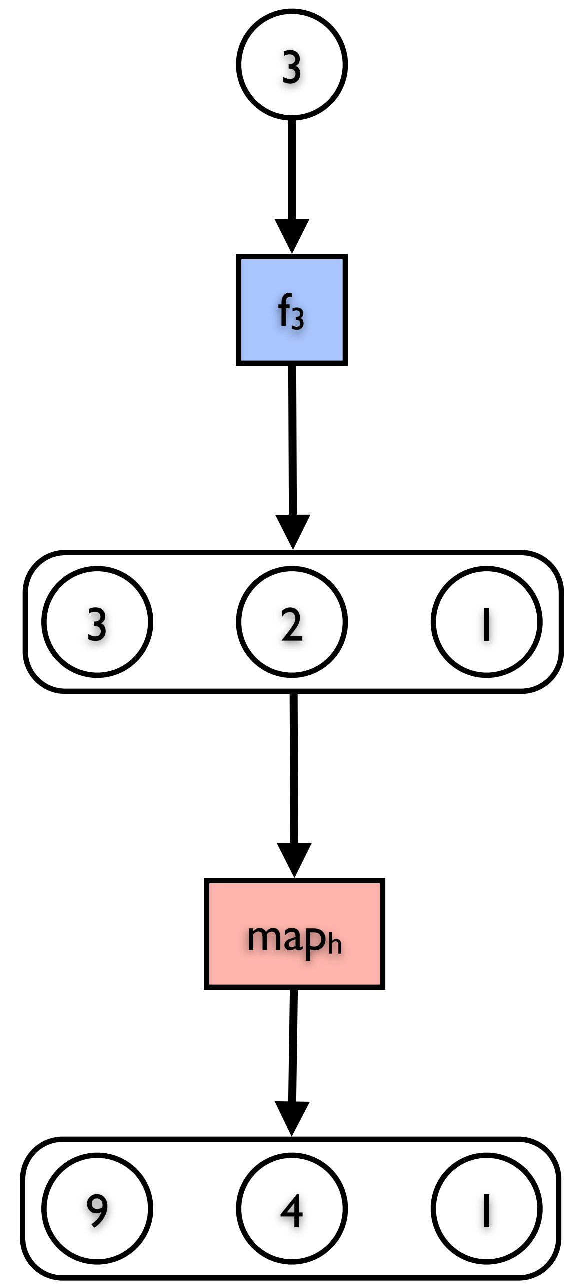
- Next we consider if-then-else conditionals.

$$e ::= \cdots \mid \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3$$

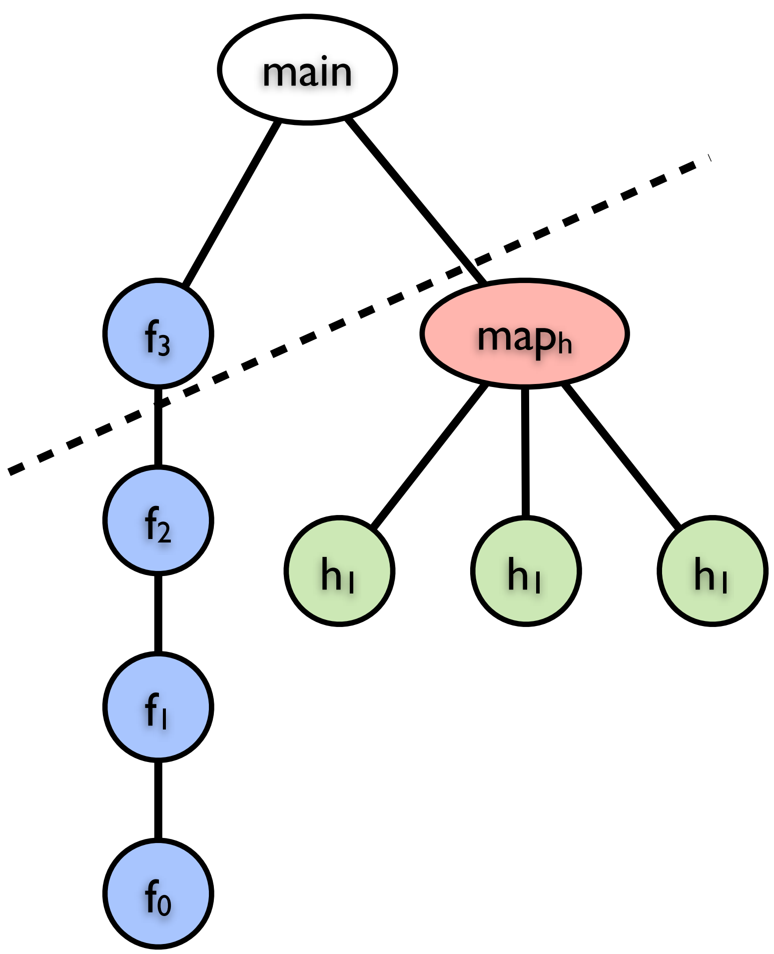
- The graph marks the conditional and direction taken.

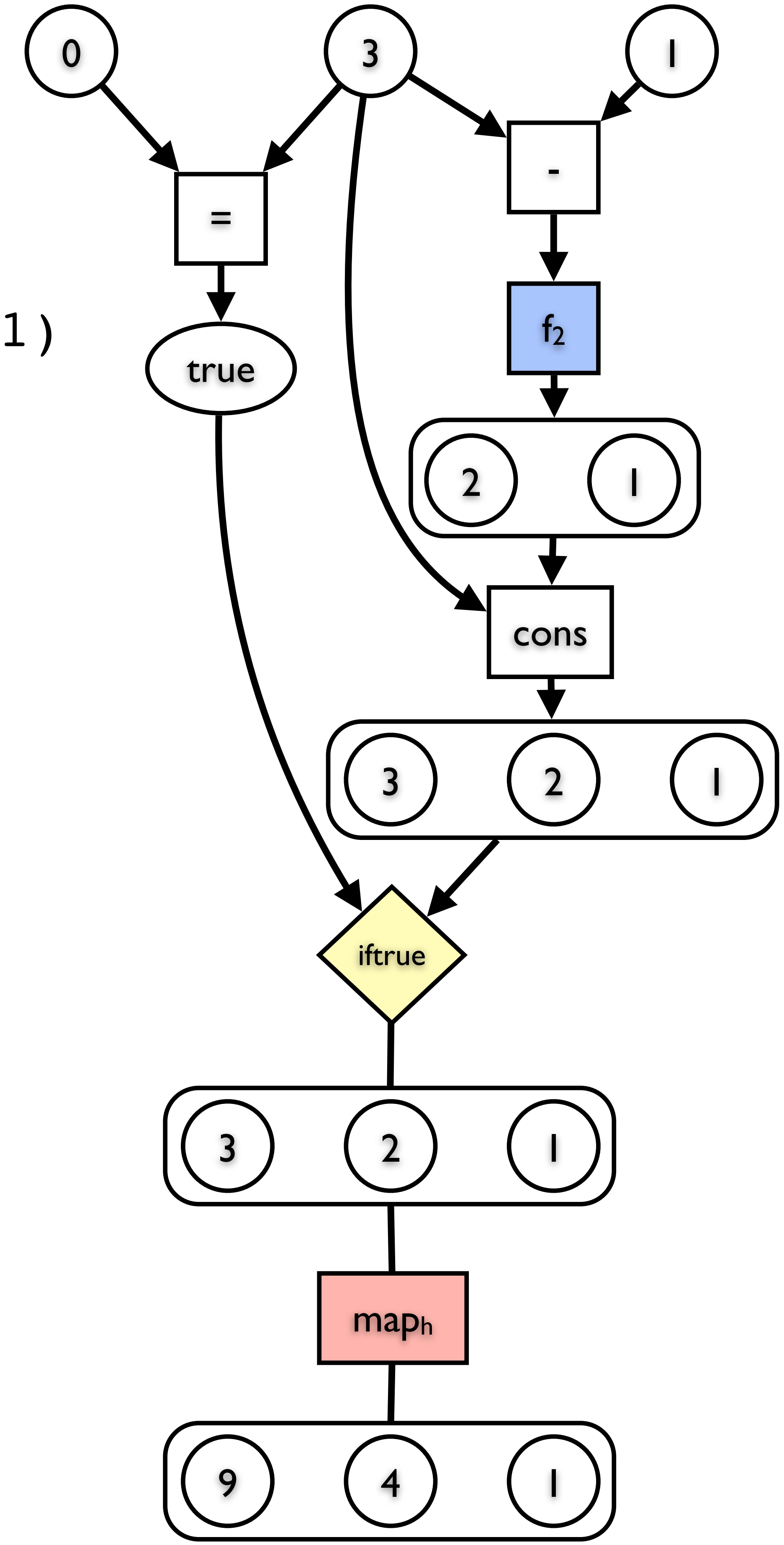
```
def f(x)    = if x = 0
              then []
              else x::f(x-1)
    h(z)    = z*z
in maph(f(3))
```





```
def f(x)    = if x = 0
              then []
              else x::f(x-1)
    h(z)    = z*z
in mapₕ(f(3))
```





Next steps

- Modeling of granularity in existing WF languages (or D-OPM)
    - expressiveness / query languages?
- Extensions to OPM / W3C PROV for hierarchical process structure?
    - complementing / clarifying work on "collections", "accounts"
- Richer language features
    - first-class higher-order functions? (cf. Perera et al. 2012)
    - meta-programming/provenance of provenance ("eval")
- Efficient implementation (exploit redundancy?)

Related work

- Provenance layering libraries/architecture (Muniswamy-Reddy et al. USENIX 2009)
- Builds on / variant of "graph model of workflow and DB prov" (Acar et al. TaPP 2010)
- ZOOM* User Views (Liu et al. TODS 2011)
    - abstract views based on user preferences
- Kepler (Anand et al. EDBT 2010)
    - data are serialized XML streams, QL supports nesting and process step navigation
- and slicing for program comprehension & provenance (Acar et al. 2012, Perera et al. 2012)
    - language-based approach, does not (directly) address abstraction/granu

Conclusions

- Provenance granularity is an important feature of several models
- There is no common understanding for what it means or how it should work
- Our contribution: basic model of "hierarchical" OPM
- But mostly open questions

- Paper/appendix gives operational semantics producing HOPM graphs

$$\frac{(a := Gen_a(c))}{\gamma, c \Downarrow \mathcal{H}(a,-,-), a} \qquad \frac{}{\gamma, x \Downarrow \mathcal{H}(-,-,-), \gamma(x)} \qquad \frac{\gamma, \vec{e} \Downarrow \vec{\mathcal{H}}, \vec{a} \qquad (a := Gen_a(\odot(\vec{a}))) \qquad (p := Gen_p(\odot))}{\gamma, \odot(\vec{e}) \Downarrow \bigcup \vec{\mathcal{H}} \cup \mathcal{H}(a, p, \vec{a}), a}$$

$$\frac{\gamma, e \Downarrow \mathcal{H}, a \qquad \gamma\{x/a\}, e' \Downarrow \mathcal{H}', a'}{\gamma, \texttt{let}\ x = e\ \texttt{in}\ e' \Downarrow \mathcal{H} \cup \mathcal{H}', a'}$$

(a)

$$\frac{\gamma, e \Downarrow \mathcal{H}, a \qquad (a^{\ell} = \texttt{true}) \qquad \gamma, e_1 \Downarrow \mathcal{H}_1, a_1 \qquad (a_n := Gen_a(a_1^{\ell})) \qquad (p_n := Gen_p(\textbf{iftrue}))}{\gamma, \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow \mathcal{H} \cup \mathcal{H}_1 \cup \mathcal{H}(a_n, p_n, (a, a_1)), a_n}$$

$$\frac{\gamma, e \Downarrow \mathcal{H}, a \qquad (a^{\ell} \neq \texttt{true}) \qquad \gamma, e_2 \Downarrow \mathcal{H}_2, a_2 \qquad (a_n := Gen_a(a_2^{\ell})) \qquad (p_n := Gen_p(\textbf{iffalse}))}{\gamma, \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \Downarrow \mathcal{H} \cup \mathcal{H}_2 \cup \mathcal{H}(a_n, p_n, (a, a_2)), a_n}$$

(b)

$$\frac{\gamma, \vec{e} \Downarrow \vec{\mathcal{H}}, \vec{a} \qquad (\gamma(f) = f(\vec{x}).e) \qquad \gamma\{\vec{x}/\vec{a}\}, e \Downarrow \mathcal{H}, a}{\gamma, f(\vec{e}) \Downarrow \bigcup \vec{\mathcal{H}} \cup \mathcal{H}_f(a, \mathcal{H}, \vec{a}), a}$$

(c)

$$\frac{\gamma, e \Downarrow \mathcal{H}, a \qquad (a^{\ell} = [c_1, \ldots, c_n]) \qquad \gamma, f(c_1) \Downarrow \mathcal{H}_1, a_1 \qquad \ldots \qquad \gamma, f(c_n) \Downarrow \mathcal{H}_n, a_n \qquad (a' := Gen_a([a_1^{\ell}, \ldots, a_n^{\ell}]))}{\gamma, \texttt{map}_f(e) \Downarrow \mathcal{H} \cup \mathcal{H}^f_{\text{map}}(a, [\mathcal{H}_1, \ldots, \mathcal{H}_n], a'), a'}$$

(d)

$$\frac{\{\vec{f}/\vec{f}(\vec{x}).\vec{e}\}, e \Downarrow \mathcal{H}, a}{\texttt{def}\ \vec{f}(\vec{x}) = \vec{e}\ \texttt{in}\ e \Downarrow \mathcal{H}_{main}(\mathcal{H})}$$