Hierarchical Models of Provenance

Peter Buneman, James Cheney, and Egor V. Kostylev
Unviersity of Edinburgh

Abstract

There is general agreement that we need to understand provenance at various levels of granularity; however, there appears, as yet, to be no general agreement on what granularity means. It can refer both to the detail with which we can view a process or the detail with which we view the data. We describe a simple and straightforward method for imposing a hierarchical structure on a provenance graph and show how it can, if we want, be derived from the program whose execution created that graph.

1 Introduction

There are numerous models of provenance [9, 7, 5] all of which provide some account of how some piece of data was derived. The reason for the variety may be partly because we collect provenance for a number of purposes (debugging, reproducibility, annotation, security etc.) and that different models are needed for these. One particularly simple model of provenance is the Open Provenance Model (OPM), which has been widely adopted for scientific workflows and other systems [9]. An OPM graph describes the causal relationships between *processes* and *artefacts*. Artefacts are data values and processes are records of some event (such as the evaluation of a function) that takes data values as inputs and produces data values as outputs. In simple cases, an OPM graph is simply a graph that describes the workflow embellished with data values. Why is this simple model not enough to capture other models or provenance? We believe that it is a reasonable starting point, but in order to do this we need to add some further structure; in particular we need to formalize hierarchical decomposition of provenance graphs.

There are several papers that have argued for the need to view provenance at various levels of granularity: OPM's accounts [9] give examples of what this might mean; ZOOM's user views [6] and Muniswamy-Reddy et al. [10] describe systems that collect or present provenance at "multiple layers of abstraction"; and [3, 2] both contain proposals for combining data and workflow provenance. What we sketch in this paper is first a formalism for imposing a hierarchical structure on an OPM-like provenance model; we then show that this structure can be derived from the execution of programs in a simple programming language that easily describes workflows; we show how, with the addition of one higher-order *map* operation we can use the same hierarchical structure to describe data granularity. We say "sketch" because some of the lengthy details of the formalism are deferred to an appendix in order to focus on basic ideas. We finally speculate on what additional structure is needed to account for other aspects of provenance such as program optimisation, the provenance of provenance graphs and invariants of provenance graphs such as semirings.

To illustrate these ideas we use a simple functional language ProvL. This language can be used to express simple workflows, branching, user-defined functions, lists, and the higher-order $\mathtt{map}_f()$ function which maps the function f to elements of a list. Two simple programs in ProvL are given in Fig. 1.

2 Hierarchical OPM graphs

Syntax and semantics of OPM graphs We start with basic OPM-style graphs without agents or accounts. Let $\mathbb{C}$ be a set of names of *constants* and $\mathbb{B}$ be a set of names of primitive (built-in) *operators* of fixed arities.

Definition 2.1 *A* OPM graph $\mathcal{G} = \langle \mathbf{A}, \mathbf{P}, \mathbf{S} \rangle$ *is an ordered labelled bipartite directed acyclic multigraph with the set of* artefact *nodes* $\mathbf{A}$ *labelled with constant names from* $\mathbb{C}$*, the set of* process *nodes* $\mathbf{P}$ *labelled with operator names from* $\mathbb{B}$*, and set of* edges $\mathbf{S}$ *such that every artefact node has one or zero outgoing "generated by" edges and every process node labelled with a operator name of arity*

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

(a)

```
let f(x) = x + 1
in map_f([3,4,5])
```

(b)

Figure 1: Programs in ProvL

n has exactly one ingoing edge and n outgoing "using" edges labelled $1,\ldots,n$.

This definition coincides with that in [9], except that we omit agents and accounts, restrict the number of ingoing edges of a process node to one and require edge labels to be numbers. These restrictions are minor and are imposed for convenience of presentation.

Next we define a semantics of OPM graphs, i.e. assign real objects to nodes of OPM graphs. For this we assume (as in Cheney [4] or Moreau [8]) that constants from $\mathbb{C}$ are interpreted as objects of arbitrary nature and operators from $\mathbb{B}$ are interpreted as functions on these objects preserving arities. We do not distinguish between names and their interpretations, and write v^ℓ for the (interpretation of the) label of a node v in an OPM graph, as well as $\vec{v}^\ell$ for the tuple of labels on a tuple $\vec{v}$ of such nodes.

Definition 2.2 *An OPM graph $\mathcal{G}$ is* valid *if for each process node p with successors $\vec{a}$ and the predecessor a we have that $p^\ell(\vec{a}^\ell) = a^\ell$.*

Fig. 2(a) shows an OPM graph representing a run of the program from Fig. 1(a). It is valid if we interpret numbers and arithmetic operations as usual.

Hierarchical OPM graphs We would like to extend OPM graphs to be able to look at them at different levels of granularity, i.e. "collapse" some parts of the graphs into single nodes when we are not interested in their details. For this we assume a set of function names $\mathbb{F}$, which are the names of functions defined in the program and which also include a top-level **main** function.

Intuitively, we enrich an OPM graph by a call tree of a run of the program (workflow) under investigation and a binding for each call in this tree of a body, input artefacts and result artefact in the graph. Formally, we have the following definition.

Definition 2.3 *An* hierarchical OPM graph*, or* HOPM graph*, is a triple $\mathcal{H} = \langle \mathcal{G}, \mathcal{T}, \mathcal{M} \rangle$, where*

1. $\mathcal{G} = \langle \mathbf{A}, \mathbf{P}, \mathbf{S} \rangle$ is an OPM graph;

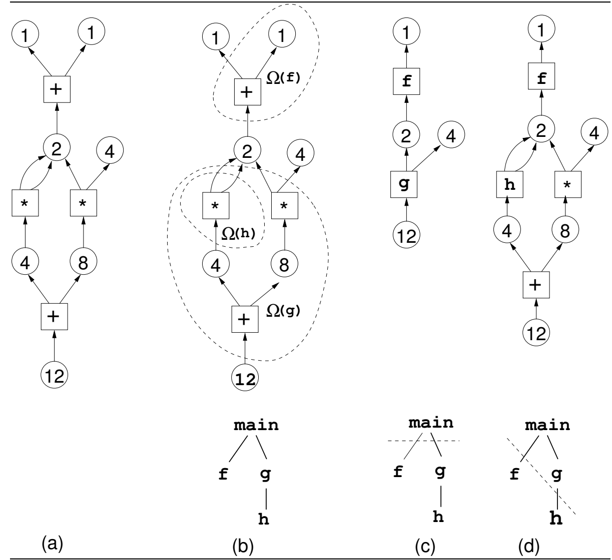

Figure 2: Hierarchical OPM graph and views

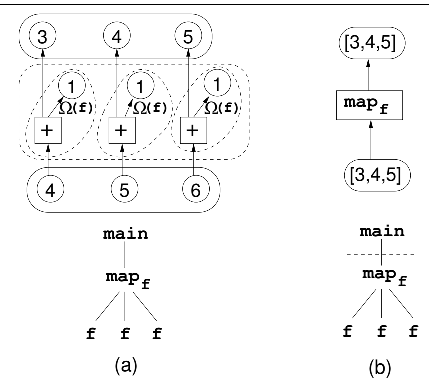

Figure 3: HOPM graph with map, and view

2. The call tree *$\mathcal{T} = \langle \mathbf{V}, \mathbf{E} \rangle$ is a directed rooted tree whose vertices $\mathbf{V}$ (referred as* calls*) are labeled with function names from $\mathbb{F}$ such that the root is labeled with* **main** *(we use s^ℓ for the label of a call s);*

3. The call mapping *$\mathcal{M} = \langle \Omega, \mathbf{in}, \mathbf{out} \rangle$, where*

- *$\Omega : \mathbf{V} \to 2^{\mathbf{A} \cup \mathbf{P}}$ is a function that associates each call in $\mathcal{T}$ with its body, i.e. a set of nodes of $\mathcal{G}$, such that:*
  - $\mathbf{P} \subseteq \Omega(\mathbf{main})$,
  - *if $(s,t) \in \mathbf{E}$ then $\Omega(s) \supseteq \Omega(t)$,*
  - *if $(s,t_1),(s,t_2) \in \mathbf{E}$ then $\Omega(t_1) \cap \Omega(t_2) = \emptyset$,*
  - *each set $\Omega(s)$ is convex,*[1] *all its ingoing edges in $\mathcal{G}$ are "generated by" edges, and all outgoing edges are "using" edges;*

[1] I.e. there is no directed path in $\mathcal{G}$ between nodes of $\Omega(s)$ which contains a node not from $\Omega(s)$.

- **in** *is a function assigning to each call s in $\mathcal{T}$ a tuple of input artefacts (maybe with repetitions) of the size of the arity of s^ℓ;*

- **out** *is a function assigning to each call s in $\mathcal{T}$ an output artefact.*

Again, we would like to give semantics to HOPM graphs. We will do it in parallel with an extension of our data model to lists. We assume that the set of constants $\mathbb{C}$ is typed, i.e. it consists of the set of primitive constants $\mathbb{C}_0$ as well as all possible nested lists over this set, including the empty list []. The set of artefacts $\mathbf{A}$ is also nested and the nesting agrees with the nesting of $\mathbb{C}$. The last means that if for $a, a_1, \ldots, a_n \in \mathbf{A}$ it holds that $a = [a_1, \ldots, a_n]$ then $a^\ell = [a_1^\ell, \ldots, a_n^\ell]$. Further, some unary function names f in $\mathbb{F}$ have corresponding mapping names $\mathtt{map}_f()$ in $\mathbb{F}$, also of arity 1.

Assume that every function name from $\mathbb{F}$ is interpreted as a function of the corresponding arity over constants from $\mathbb{C}$, and the mapping functions work as elementwise applications of the corresponding functions to lists. Again we do not distinguish a name with its interpretation.

Definition 2.4 *An HOPM graph $\mathcal{H} = \langle \mathcal{G}, \mathcal{T}, \mathcal{M} \rangle$ is* valid *if the underlying OPM graph $\mathcal{G}$ is valid and for each call s in $\mathcal{T}$ we have that*

- *if s^ℓ is a first-order function, then $\mathbf{in}(s)$ consists of the second components of all the outgoing "using" edges from $\Omega(s)$, $\mathbf{in}(s)$ is the first component of the ingoing "generated by" edge of $\Omega(s)$, and $s^\ell(\mathbf{in}(s)^\ell) = \mathbf{out}(s)^\ell$;*

- *if s^ℓ is $\mathtt{map}_f()$, then*
  - $\mathbf{in}(s)$ *has one and only one element a which is a list $[a_1, \ldots, a_n]$,*
  - $\mathbf{out}(s) = b$ *such that it is a list $[b_1, \ldots, b_m]$,*
  - *successors of s in $\mathcal{T}$ are $s_1, \ldots, s_k$ such that $s_i^\ell = f$ for every i, and $\Omega(s) = \cup_{1 \le i \le k} \Omega(s_i)$,*
  - *$n = m = k$ and for every i it holds that $\mathbf{in}(s_i)$ has a single node a_i, and $\mathbf{out}(s_i) = b_i$.*

Fig. 2(b) shows the HOPM graph version of the run of the program from Fig. 1(a), where the dotted lines show the sets $\Omega(f), \Omega(g), \Omega(h)$,[2] and the input and output functions are obvious. Fig. 3(a) does the same for the program in Fig. 1(b).

[2]Here f, g and h are labels of tree calls, so, strictly speaking, they should be replaced by the calls themselves.

Views of HOPM graphs Having HOPM graphs, it is possible to look on the underlying OPM with different granularity.

Intuitively, given a HOPM graph $\mathcal{H} = \langle \mathcal{G}, \mathcal{T}, \mathcal{M} \rangle$ and a *view subtree* $\mathcal{V}$ of the tree $\mathcal{T}$, containing the root (labeled with **main**), we can define a *view* $\mathcal{G}_\mathcal{V}$ to be an ordinary OPM graph obtained from $\mathcal{G} = \langle \mathbf{A}, \mathbf{P}, \mathbf{S} \rangle$ by expanding all of the calls in $\mathcal{V}$ and leaving the remaining calls unexpanded as new process nodes.

Formally, denote $Succ(\mathcal{V})$ the set of calls of $\mathcal{T}$ which are not in $\mathcal{V}$, but have the incoming edge starting in a call from $\mathcal{V}$. We extend the set $\mathbb{B}$ of built-in operators to $\mathbb{B}_\mathcal{V}$ with all the functions $\mathbb{F}$ that are labels of calls in the set $Succ(\mathcal{V})$. For every call s of arity n from $Succ(\mathcal{V})$ in the following construction we will use a new process node p_s with the same label as the call s. It will be connected to the rest of the OPM graph by *input* edges in_s^i for each $1 \le i \le n$, coming from p_s to the i-th element of $\mathbf{in}(s)$ and labeled by i, and by the *output* edge out_s coming from $\mathbf{out}(s)$ to p_s.

Definition 2.5 *Given an HOPM graph $\mathcal{H} = \langle \langle \mathbf{A}, \mathbf{P}, \mathbf{S} \rangle, \mathcal{T}, \langle \Omega, \mathbf{in}, \mathbf{out} \rangle \rangle$ and a view tree $\mathcal{V}$, a* view *over $\mathcal{V}$ is an OPM graph $\mathcal{G}_\mathcal{V} = \langle \mathbf{A}_\mathcal{V}, \mathbf{P}_\mathcal{V}, \mathbf{S}_\mathcal{V} \rangle$ over built-in operators $\mathbb{B}_\mathcal{V}$, such that*

- $\mathbf{A}_\mathcal{V} = \mathbf{A} \backslash \{a \mid a \in \Omega(s), s \in Succ(\mathcal{V})\}$
- $\mathbf{P}_\mathcal{V} = (\mathbf{P} \backslash \{p \mid p \in \Omega(s), s \in Succ(\mathcal{V})\}) \cup \{p_s \mid s \in Succ(\mathcal{V})\}$,
- $\mathbf{S}_\mathcal{V} = (\mathbf{S} \backslash \{(v_1, v_2) \mid v_1, v_2 \in \Omega(s), s \in Succ(\mathcal{V})\}) \cup \{in_s^1, \ldots, in_s^n, out_s \mid s \in Succ(\mathcal{V}), s^\ell \text{ is of arity } n\}$.

Fig. 2(c-d) and 3(b) illustrate different views over the programs above. Note that it makes no sense to expand a function call unless all of its ancestors in the call tree have been expanded; this is why a view is defined over a subtree of $\mathcal{T}$ rather than over an arbitrary subset.

3 Interpreting workflow programs as HOPM graphs

We have defined a small core language ProvL that can be used to describe simple workflows, where the primitive operations of the language correspond to calls to external programs rather than primitive arithmetic operations. Our semantic assigns a HOPM graph to each run of a workflow. Due to space limitations, the formal definition of the semantics is placed in an appendix; here, we describe ProvL and illustrate the provenance semantics through examples.

ProvL can be decomposed into four sublanguages, illustrating increasing complexity in the generated HOPM graph, indicated using letters (a–d) in Fig. 4.

expression	$e ::= c \mid x \mid \odot(\vec{e}) \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2$	(a)
	$\mid \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$	(b)
	$\mid f(\vec{e})$	(c)
	$\mid \texttt{map}_f(e)$	(d)
program	$\texttt{def}\ f_1(\vec{x}_1) = e_1, \ldots, f_m(\vec{x}_m) = e_m\ \texttt{in}\ e'$	

Figure 4: Syntax of ProvL

(a) ProvL_0 handles simple workflows involving constant values, primitive operations, variables, and let-binding (expressing sharing). We may take the primitive operations to be the atomic "black boxes" of any conventional workflow language (e.g. Kepler, VisTrails, Taverna, ZOOM [5]) and represent any straight-line, DAG-shaped computation using these operations as a ProvL_0 expression. The corresponding (H)OPM graph is essentially the same DAG with inverted edges.

(b) ProvL_b extends ProvL_0 with conditionals (if–then–else). The generated provenance graphs include process nodes to indicate that a conditional was evaluated, and which branch was taken. (This is similar to the approach taken in the model of [2]).

(c) ProvL_f extends ProvL_b with user-defined functions, achieving a Turing-complete language (assuming the underlying set of operators includes at least basic arithmetic). The HOPM graph can have non-trivial call trees as described above.

(d) ProvL, finally, extends ProvL_f with support for lists and the map function. The HOPM graph generated for $\texttt{map}_f()$ consists of the graphs generated for the calls $f_1, \ldots, f_n$ to f on the elements of the list, plus an edge to the input list from a process node for $\texttt{map}_f()$ itself, plus an edge to this process node from the output list node. Also, the $\texttt{map}_f()$ process node contains all of the calls to f, that is, $\Omega(\texttt{map}_f()) = \Omega(f_1) \cup \cdots \cup \Omega(f_n)$.

4 Discussion

Some questions for further work:

1. What is the relationship between our notion of views and accounts in OPM? It seems that accounts can be used to represent views, but not all accounts correspond to views (for example, accounts can provide conflicting information). How are views of HOPM graphs related to, for example, the traces and trace slicing of Acar et al. [1]?
2. How can we translate provenance queries on the full graph to queries on views? Can we identify a "best" view to answer a given query? (Similar concerns arise in ZOOM system [6], which uses user preferences to induce a clustering of basic workflow steps into groups to hide details irrelevant to the user.)
3. Our notion of validity for HOPM graphs is basic: it does not, for example, require that different calls to the same function have compatible expansions. (That is, it would be legal for one call to f to expand to $+1$ and for another to expand to $*2$.) How should validity be made more precise? Can we exactly capture the provenance expressiveness of different workflow languages?
4. Our language uses conventional abstract syntax, whereas most workflows employ a graphical notation and many have features such as concurrency or streaming that are not handled by ProvL. How is our workflow model related to existing ones [5]?

References

[1] U. A. Acar, A. Ahmed, J. Cheney, and R. Perera. A core calculus for provenance. In *POST*, number 7215 in LNCS, pages 410–429. Springer, 2012.

[2] U. A. Acar, P. Buneman, J. Cheney, N. Kwasnikowska, S. Vansummeren, and J. van den Bussche. A graph model for data and workflow provenance. In *Workshop on the Theory and Practice of Provenance*, 2010.

[3] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. In *VLDB*, 2012.

[4] J. Cheney. Causality and the semantics of provenance. In *Proceedings of the 2010 Workshop on Developments in Computational Models*, 2010.

[5] S. Davidson, S. Cohen-Boulakia, A. Eyal, B. Ludaescher, T. McPhillips, S. Bowers, M. Anand, and J. Freire. Provenance in scientific workflow systems. *IEEE Data Engineering Bulletin*, 30(4):44–50, 2007.

[6] Z. Liu, S. B. Davidson, and Y. Chen. Generating sound workflow views for correct provenance analysis. *ACM Trans. Database Syst.*, 36(1):6, 2011.

[7] L. Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2(2-3):99–241, 2010.

[8] L. Moreau. Provenance-based reproducibility in the semantic web. *J. Web Sem.*, 9(2):202–221, 2011.

[9] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. T. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. G. Stephan, and J. V. den Bussche. The open provenance model core specification (v1.1). *Future Generation Comp. Syst.*, 27(6):743–756, 2011.

[10] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor. Layering in provenance systems. In *Proceedings of the 2009 conference on USENIX Annual technical conference*, USENIX'09, pages 10–10, Berkeley, CA, USA, 2009. USENIX Association.

A Generating HOPM graphs, formally

In this section we present a toy functional language ProvL and introduce its operational semantics which produce not only the result of a computation, but also a HOPM graph for this computation. We will start with a simple workflow language and then extend it step by step to branching, user defined functions (with possible recursion) and, finally, lists.

Simple workflow language ProvL_0 We start with a simple sublanguage ProvL_0 of ProvL which contains neither branching nor user-defined functions and show how to construct provenance graphs for a run of a program in ProvL_0 . This sublanguage can be considered as a language for workflows, used in biology and other areas of science.

A syntax of expressions in ProvL_0 is given in Fig. 4(a). It is based on constants from $\mathbb{C}$ and built-in operators $\mathbb{B}$, introduced in the previous section. Also, it uses a set $\mathbb{X}$ of *variables*. Hence, an expression can be either a constant c from $\mathbb{C}$, variable x from $\mathbb{X}$, operator evaluation $\odot(\vec{e})$ over a tuple of expressions $\vec{e}$ (of the proper size), or variable definition $\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$.

The syntax of programs in ProvL_0 is given in the end of Tab. 4. However, since for now we do not allow user-defined functions, we simplify it by enforcing $m = 0$. This means that a ProvL_0 program is just an expression.

The operational semantics of expressions in ProvL_0 is given in Tab. 1(a). Here, the judgment $\gamma, e \Downarrow \mathcal{H}, a$ means that in environment γ, evaluating expression e yields an HOPM graph $\mathcal{H}$ and the artefact node a in $\mathcal{H}$ is labeled with the result value (intuitvely, showing how the result was derived). Its difference with the semantics of other languages is that the result of the evaluation of an expression is not just a constant value, but an HOPM graph, which has an artefact node without incoming edges labelled by the resulting value. However, since we do not have user-defined functions, call trees and call mappings of all HOPM graphs for ProvL_0 are empty, and further in this subsection when we say "HOPM graph" we mean just the underlying OPM part.

By the reasons above, an *environment* γ is a substitution of variables from $\mathbb{X}$ not to just values from $\mathbb{C}$, but to artefact nodes from an HOPM graph constructed before. Hence, the value of a variable in the usual sense is the label of the corresponding artefact.[3]

For every expression, the construction of the corresponding HOPM graph is done by merging HOPM graphs already constructed for subexpressions, and, in some cases, a new graph fragment $\mathcal{H}(a, p, \vec{a})$, which just links inputs by edges as expected (some inputs may be empty, which is denoted as $-$). The first two inputs of this operation are *fresh, new* nodes, created in the rule. Such nodes are denoted $Gen_v(\ell)$, where ℓ is its label. (Formally, we should be a little more careful about freshness and renaming of freshly generated nodes; we elide these details which are standard but further complicate the presentation, and use the usual union symbol $\cup$ to denote such merging.)

The semantics of a program in ProvL_0 is given in the end of Tab. 1. We will explain the general construction of $\mathcal{H}_{main}(\mathcal{H})$ later, but for ProvL_0, which does not contain any calls of user-defined functions, it constructs a simple call tree consisting only of the root labeled **main** and a call mapping binding the artefact, which is the result of the computation, to the output of **main**.

Language with branching ProvL_b Simple ProvL_0 gives us rather limited expressiveness. Next we extend it with branching, obtaining the language ProvL_b. The syntax of expressions in ProvL_b extends the syntax of ProvL_0 by an if-then-else construction, as shown in Fig. 4(b).

To define the semantics of ProvL_b we assume that $\mathtt{true} \in \mathbb{C}$. We could just introduce a new built-in operator **ifthenelse** which takes three parameters as expressions, evaluates all three, and returns either the second or the third depending whether the first is $\mathtt{true}$ or not. However, this leads to potential infinite computation, and this is not how conditionals behave in most programming languages or workflow systems. Even in the absence of recursion, expanding both the taken and non-taken branch of a conditional would lead to exponential blowup in the size of the graph. Instead, to reflect the usual lazy semantics of conditionals, where only one branch is executed depending on the test, we introduce two new built-in operators **iftrue** and **iffalse** in $\mathbb{B}$ with two parameters each. They cannot be used explicitly in the expressions, but may be labels of processes in OPM graphs. It is also required that the first parameter of **iftrue** is $\mathtt{true}$ and the first parameter of **iffalse** is not $\mathtt{true}$.

The formal semantics of ProvL_b is given in Tab. 1(b).

Language with user-defined functions ProvL_f Next we want to extend the language ProvL_b to programs with subroutines (user-defined functions), possibly recursive, and extend its semantics to produce HOPM graphs with nontrivial call trees. Once we have such a structure, we can extract $\mathcal{H}_{\mathcal{V}}$, over different views $\mathcal{V}$, as described above.

The expression syntax of such a language ProvL_f extends the syntax for language with branching ProvL_b as shown in Fig. 4(c). Here f comes from the set of the function names $\mathbb{F}$ from the previous section, and the size of the tuple $\vec{e}$ should coincide with the arity of f. The

[3] We assume that the program is well-defined, i.e. that it uses only defined variables.

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

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

(a)

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

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

(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)

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

Table 1: Semantics of ProvL

program syntax for ProvL_f in the end of Tab. 4 can now contain function definitions, i.e. it can hold that $m > 0$. Of course, the size of $\vec{x}_i$ should be equal to the arity of f_i.

The semantics of ProvL_f operates now with HOPM graphs with nontrivial call trees and call mappings. The unions of such HOPM graphs work pairwise as expected. The environment γ now has heterogeneous structure: it maps not only variables from $\mathbb{X}$ to artefacts from $\mathbf{A}$, but also function names from $\mathbb{F}$ to expressions of the form $f(\vec{x}).e$ which represent bodies of these user-defined functions. To produce valid HOPM graphs we of course require that these bodies implement the interpretations of corresponding function names.

The semantics of a function call is given in Tab. 1(c). It unions the existing HOPM graphs with new constructed graph $\mathcal{H}_f(a, \mathcal{H}, \vec{a}) = \langle \mathcal{G}, \mathcal{T}, \mathcal{M} \rangle$. This construction is straightforward: the underlying OPM graph $\mathcal{G}$ just coincides with underlying graph of $\mathcal{H}$, and the call tree $\mathcal{T}$ and the call mapping $\mathcal{M}$ extends those of $\mathcal{H}$ as expected w.r.t. the function f.

The semantics of a program in ProvL_f is given in the end of Tab. 1. It forms the original environment γ with defining function bodies and evaluates the main expression. The resulting HOPM graph $\mathcal{H}$ is then enriched by the root of the call tree (labeled with **main**) and corresponding call mapping in $\mathcal{H}_{main}(\mathcal{H})$.

Language with support for lists ProvL Finally, we show how to extend the languages described before with *nested lists* and *map* function. The same can be done for other typed structures like sets or graphs and corresponding higher-order functions. For example, it can be done for HOPM graphs and programs as structures, and optimisers and evaluators as higher-order functions.

In our language ProvL, extending ProvL_f with support for lists and maps, we assume that the set of constants $\mathbb{C}$ is typed as described before. We may assume that the set of built-in operators $\mathbb{B}$ contains special functions manipulating lists: $[c]$, which creates a list of single element, $c_1 \cdot c_2$, which concatenates two lists, $\texttt{flatten}(c)$ which flattens a list, $\texttt{first}(c)$ which segregates the first element of a list, and $\texttt{rest}(c)$ which removes the first element from a list. Next we implicitly assume that all expressions are well-typed, i.e. these built-in operators indeed have lists as parameters.

The syntax of expressions in ProvL, extends ProvL_f with the higher-order function $\texttt{map}_f(e)$, and is given in Fig. 4. In the rule (d) f is a user-defined function from $\mathbb{F}$ of arity 1. As above, we assume that this parametrized by f function $\texttt{map}_f(e)$ belongs to the set of functions $\mathbb{F}$. Its semantics is shown in Tab. 1(d). Intuitively, it evaluates the body of f with the variable x substituted with each of the elements of the list e, and composes the list of results. It unions the HOPM graph $\mathcal{H}$, which is the result of the evaluation of e, with a new HOPM graph

$$\mathcal{H}^{f}_{\mathrm{map}}(a, [\mathcal{H}_1, \ldots, \mathcal{H}_n], Gen_a([a^{\ell}_1, \ldots, a^{\ell}_n])).$$

This graph links the input artefact a labelled with the list $[c_1, \ldots, c_n]$ with created in computation of $\mathcal{H}_1, \ldots, \mathcal{H}_n$ artefacts $Gen_a(c_1), \ldots, Gen_a(c_n)$ labelled with its elements. Also, it links the results $a_1, \ldots, a_n$ of the function calls with $Gen_a([a^{\ell}_1, \ldots, a^{\ell}_n])$. Finally, it extends the call tree and the call mapping as expected.