7

DOCTORAATSPROEFSCHRIFT

2011 |School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

Dataflows and Provenance: From Nested Relational Calculus to the Open Provenance Model

Proefschrift voorgelegd tot het behalen van de graad van Doctor in de Wetenschappen, Informatica, te verdedigen door:

Natalia KWASNIKOWSKA

Promotor: prof. dr. Jan Van den Bussche

Maastricht University

universiteit hasselt

DOCTORAATSPROEFSCHRIFT

2011 |School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

Dataflows and Provenance: From Nested Relational Calculus to the Open Provenance Model

Proefschrift voorgelegd tot het behalen van de graad van
Doctor in de Wetenschappen: Informatica, te verdedigen door:

Natalia KWASNIKOWSKA

Promotor: prof. dr. Jan Van den Bussche

Maastricht University

universiteit
hasselt

D/2011/2451/31

Acknowledgments

First and foremost, I would like to thank my advisor, Jan Van den Bussche, for his guidance, support and encouragement throughout the research reported here. I would also like to thank Deborah Dumont, Anna Gambin, Jan Hidders, Kerstin Koch, Sławomir Lasota, Lennart Martens, Luc Moreau, Jean-Paul Noben, Jacek Sroka, and Jerzy Tyszkiewicz, all of whom I had the pleasure to collaborate with in the context of this thesis.

I would also like to mention my appreciation for Yi Chen and Zoé Lacroix, with whom I collaborated on a related topic.

Special thanks to Frank Neven, Geert Jan Bex, Tim Dupont, Steven Engels, Wouter Gelade, Goele Hollanders, Kerstin Koch, Dirk Leinders, and Dieter Van de Craen, for lessons learned when teaching various subjects.

I would like to thank the members of InfoLab, and especially Marc Gyssens, for providing a stimulating working environment.

Many thanks to Geert Jan Bex, Kerstin Koch, Dirk Leinders and Vania Bogorny, for frequently providing services as a sounding board.

I would like to express my gratitude to Bash, Geert Jan Bex, Magda Bex, Guy Van Diest, and Jan Van den Bussche: I am indebted to them, as well as to all medical professionals involved, for their assistance and support during a difficult period.

Finally, I would like to thank my family and friends for their support. Last but not least, I would like to thank Jan, Chris, Lien, Maarten, and Viviane, for taking care of Bash.

Ghent, June 2011

Contents

1 Introduction

A workflow is a high-level description of a complex and possibly long-during process, consisting of different tasks, that must be performed in a specified order [vdAvH04, SGBB01, BF05, FKSS08]. Tasks are composed of other tasks, implying control flow dependencies. The control flow need not be linear: some tasks can be performed concurrently, or alternatively. In addition to the control flow, there is also data flow. One distinguishes between a workflow *specification*, also known as a workflow *template*, or a *process definition*, or an *executable workflow* (depending on the supported level of abstraction), and the actual *executions* of a workflow specification. In an enterprise setting, there may be many different workflow specifications in use, and every specification may have been executed many times. A *workflow management system* facilitates the design of workflow specifications, and management of the different executions.

Workflow management [vdAvH04] has its origins in business process modelling, but in recent years workflows have gained importance in e-Science, in parallel with the rise of Grid Computing [FK04]. Although a strong separating line between business workflows and scientific workflows is impossible to draw, there is a clear tendency in scientific workflows to be less focused on control flow, and more focused on data flow [SKDN05, GDE$^+$07, FKSS08]. For this reason, we refer to scientific workflows as *dataflows*.

The use of workflows in e-Science includes the following aspects:

- design and execution of dataflows;
- storage of dataflows and related data in a repository;
- querying of a dataflow repository;

- sharing of dataflow data with others.

Dataflow repositories, in particular, can serve many important purposes:

- Effective management of all experimental and workflow data that float around in a large laboratory or enterprise setting, helping to "enforce the scientific method" [Bro08].
- Verification of results, either within the laboratory, by peer reviewers, or by other scientists who try to reproduce the results. Verification often involves tracking the provenance (origin) of (parts of) data values occurring in the output of a dataflow execution.
- Making all data and stored dataflows available for complex decision support or management queries. The range of such possible queries is enormous; just to give two examples, we could ask "Did an earlier run of this dataflow, using an older version of GenBank, also have this gene as a result?"; or "Did we ever run a dataflow in which this sequence was also used in a BLAST search?".

A formal model of dataflow repositories The first goal of our work is to contribute a formal, conceptual data model that synthesises some key database aspects of dataflow management. Many workflow management systems and provenance management systems[*] have already been developed, often on top of general-purpose databases.[†] The database systems used can be (i) key-value [HSBMR08, MGM$^+$08], (ii) relational [CM95, AIL98, SKDN05, BD08, GCM$^+$11, FSC$^+$06, SPG08, KDG$^+$08, MGM$^+$08], (iii) XML [FSC$^+$06, SPG08, FMS08], (iv) RDF [KDG$^+$08, ZGST08], or (v) file-based [LAB$^+$06, MGM$^+$08].

There is, however, no standard underlying conceptual data model specific to such systems. Such a data model should cover at least the following aspects:

- A data model for the complex data structures that are given as input to a dataflow, or that are produced as output, or that occur as intermediate results.

[*] In the context of workflows, a provenance management system is a system that only captures information about workflow executions. They allow for use of scripts, general purpose programming languages, and higher-level workflow languages alike.

[†] Some systems use different database systems for workflow specifications on the one hand, and workflow executions on the other.

- A formal operational semantics for operators, provided by the workflow system, performing basic transformations on the complex data structures.
- A formal definition of the information present in the execution of a dataflow. This information is often called the *workflow provenance* [CT09, MCF+11] or *retrospective provenance* [FKSS08] of the data output of the workflow.
- The involvement in dataflow executions of external services, typically supplied by a third party, for which the dataflow system provides a wrapper. These external services form the basic tasks of a scientific workflow.
- The use of a dataflow serving as a task in another dataflow.
- A conceptual model for a repository consisting of many dataflow specifications and their executions.
- The querying of the entire information present in such a dataflow repository. Of particular interest are queries for the finer-grained provenance of parts of data output of workflows. This kind of queries are often called *where-provenance*.

A conceptual data model for dataflow repositories should offer a precise specification of the types of data (including the dataflows themselves) stored in the repository, and of the relationships among them. Such a data model is important because it provides a formal framework that allows:

- Analysing, in a rigourous manner, the possibilities and limitations of dataflow repositories.
- Comparing, again in a rigourous manner, the functionality of different existing systems.
- Highlighting differences in meaning of common notions as used by different authors or in different systems, such as "workflow", "provenance", or "collection".

It should be clear that the purpose of our contribution is *not* to propose a blueprint for a new dataflow management system with innovative features that "competing" systems do not support yet. Rather, our work is a detailed effort to model such a system, hoping to contribute a formally defined synthesis of some key database aspects of dataflow management systems. Indeed, each of the aspects we touch upon has been addressed in existing systems, of course each particular system with its own emphasis. In a perfect world, one would like a standard database schema for the exchange of dataflow specifications and executions, similar to the myExperiment.org initiative that is specific to the Taverna system. The need for such a repository has also been emphasised by Lacroix [KCL06, KLL$^+$07]. Our work is a new step in this direction, after initial steps during the 1990s [CM95, AIL98] did not receive the follow-up they deserved. The need for a workflow repository is also acknowledged in other fields, as shown by Blockeel and Vanschoren's Experiment Databases for Machine Learning [Blo06, BV07].

Dataflow management systems have been largely developed within the computer systems community, and have received less interest in the database theory community. We hope to partially fill this gap by the present work. We do note that much attention has been paid to automated verification of data-intensive workflows, but this is a research focus that is orthogonal to the focus on data modelling and querying taken in this work.

For a review of scientific workflow management and provenance systems, we refer to Freire et al. [FKSS08], Yogesh et al. [SPG05], Bose and Frew [BF05], and Davidson and Freire [DF08]. Where-provenance is one form of data provenance as investigated in database research [BKT01, CT09, CCT09]. In this work we show how the concept of where-provenance can also be defined in the context of workflow provenance.

Semantics for the Open Provenance Model The second goal of our work is to contribute to the semantics of the Open Provenance Model (OPM) [MCF$^+$11]. OPM is a proposed system-independent format for exchanging provenance data. OPM responds to the same need of a standard data model for workflows addressed above, but OPM targets past executions/activities of diverse processes (which need not be workflows as understood in business or e-Science), and data dependencies occurring in these executions. In a quest to understand emerging models for provenance in Semantic Web technologies, the W3C Provenance Incubator Group[‡] decided to map the concepts employed in these models to a single target model. OPM has been adopted as this target model. Very recently, a W3C Provenance Working Group[§] has been formed, with the goal of defining a standard language for exchanging provenance information, proposed earlier by Moreau [Mor10], and based on the review and

[‡]`http://www.w3.org/2005/Incubator/prov/wiki/Main_Page`

[§]`http://www.w3.org/2011/prov/wiki/Main_Page` roadmap provided by the W3C Provenance Incubator Group.

The current specification of OPM [MCF$^+$11], however, is purely syntactical. Yet, a semantics is suggested, both informally, and in the form of a number of inference rules. We present a formal, temporal semantics for OPM, and give a complete set of inference rules for temporal inference in OPM. This work has been done in collaboration with Luc Moreau [KMV10].

There have already been other efforts to provide semantics for OPM.

Cheney [Che10] investigates the use of structural causal models as a semantics for provenance graphs, and relates some OPM concepts to notions of actual cause and explanation proposed by Halpern and Pearl [HP05].¶ The temporal semantics we propose is similar to Cheney's in the sense that it provides a mathematical meaning for OPM graphs. However, it differs in other significant ways: (i) our semantics conforms to the OPM reference specification [MCF$^+$11] and in particular handles time, all permitted OPM edges, algebraic operations and refinements; Cheney's semantics regards single-step derivation edges as inferable, when they can only be asserted in OPM, and does not characterise inferred edges; (ii) our semantics is purely temporal, and does not see an OPM graph as a function operating on some inputs and generating outputs; this view of OPM graphs is complementary to ours, but relies on meanings associated with processes by means of annotations, which have not been modelled here. (iii) Cheney's semantics attempts the more ambitious goal of providing a global approximation (using the predictive nature of causal models) for the program being executed (without having its explicit code), so that its behaviour can be repeated for any arbitrary input; our semantics is agnostic about the predictive nature of OPM graphs.

Missier and Goble [MG11] address the question of whether, for any OPM graph, a plausible workflow exists in the Taverna workflow language, which could have generated the graph. To this end, they identify the extra information that should be captured as part of an OPM graph so that the mapping from OPM to a workflow representation can be derived. Thus, this work derives a trace semantics for OPM, obtained by composing their translation and the Taverna semantics [SH09a, SH09b, SHMG10]. It however does not tackle OPM in full, ignoring time and refinement; their translation should be revisited to leverage the distinction between precise and imprecise edges, introduced in this work.

¶See Note A.4.12 in the Appendix for a more detailed explanation of causality in OPM.

1.1 Chapter Overview

In Chapter 2, we define our dataflow model for complex objects [HKS+07, HKS+08]. Indeed, our model is strongly based on the well-known complex object data model. As the language in which to write dataflow specifications, we naturally adopt the Nested Relational Calculus (NRC). This language serves as an abstraction for basic operations on complex objects within a dataflow. Such operations can be provided by the workflow system as tasks that are useful for conversion between data formats, or for formatting data for presentation.

We note that several workflow management systems already support complex data, e.g., Taverna [TMG+07, MBZ+08], Taverna 2 [SHMC10], Kepler-CoMaD [MBL06, BML08], and Pegasus-Chimera [FVWZ02, CFV+08]. The operation of applying a function on all elements of a collection is typically provided.

However, in a dataflow, many operators can be used to connect the basic tasks. In our work, we focus on those operators that perform complex object transformations, and here we restrict attention to the basic and well-established set of operators provided by NRC. Therefore, we use NRC, extended with tasks, as a dataflow specification language.

The suitability of NRC (in the form of a variant language called CPL) for scientific data manipulation and integration purposes has already been amply demonstrated by the Kleisli system [CCW03, DW04]. We have confirmed this further by doing some case studies ourselves (including dataflows based on published bioinformatics protocols [GJZ06, ŁKO+06, NDK+06, GV04]).

In Chapter 3, we define a conceptual model for a dataflow repository.

- We formally define executions, or *runs* as we call them, of NRC dataflows.
- We formalise the representation of external services in a dataflow repository.
- We formalise the substitution of abstract tasks in a dataflow specification, either by external services, or by calls to other dataflows.
- We present a simple conceptual schema of a dataflow repository, and formalise some essential integrity constraints.
- We describe a proof-of-concept representation in SQL:2003.

In Chapter 4, we discuss the querying of the information present in a dataflow repository. Queries can, in general, be programmed in SQL/XML. Nevertheless, we identify and formally define *subvalue provenance* as a useful query primitive. For any subvalue of the final result of any run, this function computes the where-provenance of that subvalue. Moreover, we identify the *subruns* of a run. This allows us to compute the where-provenance of subvalues of intermediate results, by using the corresponding subrun.

We note that many workflow management systems already use standard query languages for querying their repository [MLA+08, FKSS08, SGM11]:

- REDUX [BD08], Swift [GCM+11] and Pegasus [KDG+08] use SQL,
- Taverna [ZGST08] and Pegasus use SPARQL, and
- ES3 [FMS08] and PASOA/PreServ [MGM+08] use XQuery.

Unfortunately, all these systems use different logical database schemas; exchange of information can only be done using the common OPM format.

We should also mention some more distantly related work. Beeri et al. are working on an interesting project on querying the *potential* executions of a given workflow specification [BEKM08]. That approach is mainly verification-oriented rather than repository-oriented, although they did also consider monitoring [BEMP07].

Finally, in Chapter 5, we present a formal, temporal semantics for the Open Provenance Model [MCF+11]. Our main contribution consists of a set of inference rules that work entirely on the logical level of OPM graphs, yet capture in a sound and complete sense all temporal inferences that can be drawn from the temporal axioms associated with an OPM graph.

2

Dataflow model

2.1 Complex values

In this section we formally define *complex values*, which we use to model complex data structures used in dataflows. Complex values are constructed, using record and set constructions, from *base values*. Base values can be numbers, strings, or booleans, but can also be flat files or XML files, in essence any value that is considered atomic in a dataflow.

Basically, the designer of a scientific workflow decides which kinds of values are considered to be atomic, and for which kinds of values he needs to explicitly model the internal structure. The latter is only necessary if the complex values need to be manipulated by operators in the dataflow. It is the responsibility of the designer to choose which data manipulation aspects need to be explicitly modelled, and which can be modelled as a single step: a good formal model should not enforce this choice in a particular direction.

Definition 2.1 (Complex values). Assume a countably infinite set $\mathcal{A}$ of *base values* and a countably infinite set $\mathcal{L}$ of *labels*, with $\mathcal{A} \cap \mathcal{L} = \emptyset$. We define the set $\mathcal{V}$ of *complex values* as the smallest set satisfying the following:

- $\mathcal{A} \subseteq \mathcal{V}$,
- if $v_1, \ldots, v_n \in \mathcal{V}$, then the finite set $\{v_1, \ldots, v_n\}$ is a complex value,
- if $v_1, \ldots, v_n \in \mathcal{V}$, and $l_1, \ldots, l_n \in \mathcal{L}$ are distinct labels, then the named tuple $\langle l_1 \colon v_1, \ldots, l_n \colon v_n \rangle$ is also a complex value. The positioning of elements $l_i \colon v_i$, for $i \in \{1, \ldots, n\}$, within a named tuple is arbitrary.

From now on we abbreviate "named tuple" to "tuple".

Note that we work with sets as the basic collection type, because other kinds of collections can be modeled as sets of records. For an ordered list, for example, one could use a numerical attribute that indicates the order in the list.

Next we illustrate the use of complex values with a simple example from bioinformatics.

Example 2.2. Let $\mathcal{A}$ include the following set:

$$\{\texttt{EF051731.fasta}, \texttt{EF051731mRNA.fasta}, \texttt{FASTA}, \texttt{EF051731}, \texttt{ABK79072}, \\ \texttt{hemoglobin delta gene}, \texttt{DNA}, \texttt{RNA}, \texttt{homo sapiens}\}\,,$$

and let

$$\{\mathit{annotation}, \mathit{desc}, \mathit{file}, \mathit{format}, \mathit{id}, \mathit{organism}, \mathit{proteins}, \\ \mathit{sequences}, \mathit{type}, \mathit{nseq}\}$$

be a subset of $\mathcal{L}$. Base values EF051731.fasta and EF051731mRNA.fasta are place holders for the actual content of the corresponding flat files in FASTA-format. Value EF051731.fasta contains information about the gene with accession number EF051731, and EF051731mRNA.fasta contains information about its corresponding messenger-RNA sequence. If we want to analyse the nucleotide sequence of that gene, we may need different information at different steps of the analysis. If we want to perform a BLAST-search, and some program expects the whole content of a FASTA flat file, we can pass base value EF051731.fasta as input to that program. However, if that program expects only a nucleotide sequence, we can make the following complex value:

$$\mathit{gene} = \langle \mathit{id} : \texttt{gi|118199992|gb|EF051731.1|}, \\ \mathit{desc} : \texttt{homo sapiens} \ldots \texttt{complete cds}, \mathit{nseq} : \texttt{GCAGAGT} \ldots \texttt{AGGCTCT}\rangle\,,$$

and subsequently use *gene.nseq* as input value to that program. Note that we can populate the set of base values according to our needs.

We can also use complex values for bundling relevant information. For example, if we want to look for non-transcribed regions in that gene, we may need to construct the following complex value:

$$\mathit{seqs} = \{\langle \mathit{type} : \texttt{DNA}, \mathit{format} : \texttt{FASTA}, \mathit{file} : \texttt{EF051731.fasta}\rangle\,, \\ \langle \mathit{type} : \texttt{RNA}, \mathit{format} : \texttt{FASTA}, \mathit{file} : \texttt{EF051731mRNA.fasta}\rangle\}\,.$$

If we are collecting annotation information about that gene, we may want to work with the following:

$$ann = \langle id: \texttt{EF05173}, organism: \texttt{homo sapiens}, proteins: \{\texttt{ABK79072}\},$$
$$desc: \texttt{hemoglobin delta gene}\rangle.$$

Of course, we can also put all available information into a single complex value:

$$geneData = \langle annotation: ann, sequences: seqs\rangle.$$

□

2.1.1 Subvalue path of a complex value

Intuitively, a *subvalue* of a complex value v is a complex value occurring in v, possibly v itself. However, a subvalue may have different occurrences. For example, in $\{\langle id: \texttt{1},\ set: \{\texttt{1},\texttt{2},\texttt{3}\}\rangle\}$, subvalue 1 appears twice. We need a formal way to distinguish these occurrences. Therefore we introduce the notion of a *subvalue path*.

A subvalue path of a complex value v is a sequence over $\mathcal{V} \cup \mathcal{L}$,* starting with the value v itself and containing subvalues of v. If a complex value in the sequence is not the final subvalue of the path, then it is followed by one of its elements if it is a set, or by an element label and its corresponding value if it is a tuple.

Formally, we define a subvalue path of a complex value by the rules given in Definition 2.3. We use symbol "φ" to denote a subvalue path, and notation $\varphi \leftarrow\!\!\bullet\ v$ for "φ is a subvalue path of v".

When a subvalue path φ of v is not $[v]$ itself, we call it a *proper subvalue path* of v. We then use $\varphi \leftarrow\!\!\dot{\bullet}\ v$ as notation. We frequently address the subvalue path $[v]$ of v as the *trivial subvalue path*.

Definition 2.3 (Subvalue path of a complex value).

$$\frac{v \in \mathcal{V}}{[v] \leftarrow\!\!\bullet\ v} \qquad \frac{v = \{v_1, \ldots, v_n\} \qquad \varphi \leftarrow\!\!\bullet\ v_i \qquad i \in \{1, \ldots, n\}}{[v] \cdot \varphi \leftarrow\!\!\bullet\ v}$$

$$\frac{v = \langle l_1: v_1, \ldots, l_n: v_n\rangle \qquad \varphi \leftarrow\!\!\bullet\ v_i \qquad i \in \{1, \ldots, n\}}{[v; l_i] \cdot \varphi \leftarrow\!\!\bullet\ v}$$

*Sequences are formally defined in Section A.1

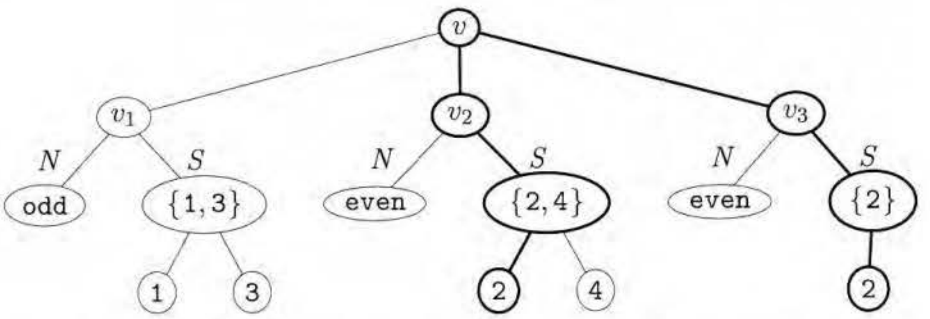


Figure 2.1: Tree-representation of value $v = \{v_1, v_2, v_3\}$ from Example 2.4. Note the two occurrences of subvalue 2 and their corresponding subvalue paths.

Example 2.4. Let v be the set $\{v_1, v_2, v_3\}$ with $v_1 = \langle N\colon \texttt{odd}, S\colon \{\texttt{1},\texttt{3}\}\rangle$, $v_2 = \langle N\colon \texttt{even}, S\colon \{\texttt{2},\texttt{4}\}\rangle$, and $v_3 = \langle N\colon \texttt{even}, S\colon \{\texttt{2}\}\rangle$, then v has the following subvalue paths:

$[v]$	$[v, v_2]$	$[v, v_3]$
$[v, v_1]$	$[v, v_2, N, \texttt{even}]$	$[v, v_3, N, \texttt{even}]$
$[v, v_1, N, \texttt{odd}]$	$[v, v_2, S, \{\texttt{2},\texttt{4}\}]$	$[v, v_3, S, \{\texttt{2}\}]$
$[v, v_1, S, \{\texttt{1},\texttt{3}\}]$	$[v, v_2, S, \{\texttt{2},\texttt{4}\}, \texttt{2}]$	$[v, v_3, S, \{\texttt{2}\}, \texttt{2}]$
$[v, v_1, S, \{\texttt{1},\texttt{3}\}, \texttt{1}]$	$[v, v_2, S, \{\texttt{2},\texttt{4}\}, \texttt{4}]$	
$[v, v_1, S, \{\texttt{1},\texttt{3}\}, \texttt{3}]$		

Note that there are actually two different subvalue paths ending in subvalue 2, i.e., $[v, v_2, S, \{\texttt{2},\texttt{4}\}, \texttt{2}]$ and $[v, v_3, S, \{\texttt{2}\}, \texttt{2}]$. We see that those subvalue paths identify different occurrences of subvalue 2 in v. □

2.1.2 Tree-representation of a complex value

It is convenient to represent a complex value as a tree, where each node is labelled with the subvalue it represents. If such a subvalue is a set, then each child node represents one element of that subvalue, and its connecting edge is unlabelled. If such a subvalue is a tuple, then each child node represents the complex value of one element of that subvalue, and its connecting edge is marked by the corresponding element label. In Figure 2.1 we see the tree-representation of value $v = \{v_1, v_2, v_3\}$ from Example 2.4. Note that a subvalue path of a complex value v corresponds to a simple path in the tree representation of v, starting at the root and ending in one of its nodes.

2.2 Complex types

Types are a basic mechanism in computer programming used to avoid the application of operations to inputs on which the operation is not defined. All data occurring in our dataflow model are strongly typed. We use a typing system for complex objects with tuples and sets, as defined by Pierce ([Pie02]), that includes a form of sub-typing.

Definition 2.5 (Complex types). Assume a finite set $\mathcal{B}$ of *base types*, and a symbol "$\perp$" that is not in $\mathcal{B}$. We define the set $\mathcal{T}$ of *complex types* as the smallest set satisfying the following:

- $\perp \in \mathcal{T}$,
- $\mathcal{B} \subseteq \mathcal{T}$,
- if $\tau \in \mathcal{T}$, then the expression $\{\tau\}$ is also a complex type, called a *set type*,
- if $\tau_1, \ldots, \tau_n \in \mathcal{T}$, and $l_1, \ldots, l_n \in \mathcal{L}$ are distinct labels, then the expression $\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$ is also a complex type, called a *tuple type*. The positioning of elements $l_i : \tau_i$, $i \in \{1, \ldots, n\}$, within a tuple type is arbitrary.

For convenience, we abbreviate "complex type" to "type".

We use "$\perp$" to denote a special *bottom type*, the purpose of which is to have a generic type for the empty set — that type is the set type $\{\perp\}$.[†] The purpose of base types is obviously to organise the base values in classes. Generally, the semantics of types is that for each type we have a set of values of that type.

Definition 2.6 (Type semantics). We define the *set of values of type* τ, denoted by $[\![\tau]\!]$, as follows:

- $[\![\perp]\!] = \emptyset$,
- $[\![\mathbf{b}]\!]$, with $\mathbf{b} \in \mathcal{B}$, is the set of base values of type $\mathbf{b}$, i.e., a non-empty subset of $\mathcal{A}$ determined by the application,
- $[\![\{\tau\}]\!]$ is the set of all finite subsets of $[\![\tau]\!]$,
- $[\![\langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle]\!]$ is the set of all tuples $\langle l_1 : v_1, \ldots, l_n : v_n \rangle$, where $v_i \in [\![\tau_i]\!]$, $i \in \{1, \ldots, m\}$, and $m \leq n$.

[†]Type $\{\perp\}$ for the empty set is used in Section 2.4.2.

Note that we regard a tuple type $\langle l_1 : \tau_1, \ldots, l_m : \tau_m \rangle$ as describing all tuples with *at least* all elements l_i of type τ_i, for $i \in \{1, \ldots, m\}$. This definition conforms to the definition by Pierce [Pie02].

We assume that at least the base type **Boolean** is always an element of $\mathcal{B}$ with $[\![\mathbf{Boolean}]\!] = \{\texttt{true}, \texttt{false}\}$.

2.2.1 Subtyping relation

Reasons of flexibility require that a type system used for scientific workflows is equipped with a form of subtyping [Pie02]. Base types provide an organisation of the different types of base values into different classes, and it is standard practice to allow for classes and subclasses. Moreover, subtyping allows a flexible typing of if-then-else statements in dataflows. Thus, the type system of our dataflow model, while guaranteeing safe execution of operations, does not impede flexible specification of dataflows.

Definition 2.7 (Subtyping relation)**.** We assume a *subtyping relation* on the set $\mathcal{B}$ of base types as a partial order relation $\preceq$ on $\mathcal{B}$ that satisfies the following condition:

$$\forall\, \mathbf{b_1}, \mathbf{b_2} \in \mathcal{B} \colon \mathbf{b_1} \preceq \mathbf{b_2} \implies [\![\mathbf{b_1}]\!] \subseteq [\![\mathbf{b_2}]\!]. \tag{2.1}$$

This relation can be arbitrarily defined by the application, as long as condition (2.1) is satisfied.

A subtyping relation $\preceq$ on $\mathcal{B}$ is canonically extended to the set $\mathcal{T}$ of complex types as follows:

- $\bot \preceq \tau$, for any complex type τ.
- a set type can only be a subtype of another set type. For two set types $\tau = \{\tau'\}$ and $\mu = \{\mu'\}$, we have $\tau \preceq \mu$ if and only if $\tau' \preceq \mu'$.
- a tuple type can only be a subtype of another tuple type. For two tuple types $\tau = \langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle$ and μ, we have $\tau \preceq \mu$ if and only if μ can be written as $\langle l_1 : \mu_1, \ldots, l_m : \mu_m \rangle$, with $m \leq n$, and $\tau_i \preceq \mu_i$ for $i \in \{1, \ldots, m\}$.

We read $\tau \preceq \mu$ as "τ is a subtype of μ" or "μ is a supertype of τ". Note that if τ is a base type and $\tau \preceq \mu$, then μ is also a base type.

Next we illustrate the use of complex types and subtyping with a simple example from bioinformatics.

$$\begin{tikzcd}
 & \textbf{String} & & & \textbf{BioSeq} & \\
\textbf{ID} \arrow[ur] & \textbf{Locus} \arrow[u] & \textbf{Description} \arrow[ul] & & \textbf{NucleotideSeq} \arrow[u] & \textbf{AminoAcidSeq} \arrow[ul] \\
\textbf{GeneID} \arrow[u] & \textbf{ProteinID} \arrow[ul] & & \textbf{DNA} \arrow[ur] & \textbf{mRNA} \arrow[u] & \textbf{PeptideSeq} \arrow[u]
\end{tikzcd}$$

Figure 2.2: Partial order relation for $\mathcal{B}$ in Example 2.8

Example 2.8. Let $\mathcal{B}$ include the following set:

$$\{\textbf{AminoAcidSeq}, \textbf{BioSeq}, \textbf{Description}, \textbf{DNA}, \textbf{GeneID}, \textbf{ID}, \textbf{Locus}, \\ \textbf{mRNA}, \textbf{NucleotideSeq}, \textbf{PeptideSeq}, \textbf{ProteinID}, \textbf{String}\},$$

and let

$$\{\mathit{desc}, \mathit{gene}, \mathit{id}, \mathit{loc}, \mathit{mRNA}, \mathit{pept}, \mathit{prot}, \mathit{seq}\}$$

be a subset of $\mathcal{L}$. Figure 2.2 shows us a partial order relation defined on $\mathcal{B}$. Let us consider the following tuple types:

$$\text{BioSequence} = \langle \mathit{id}: \textbf{ID}, \mathit{desc}: \textbf{Description}, \mathit{seq}: \textbf{BioSeq}\rangle$$

$$\text{Gene} = \langle \mathit{id}: \textbf{GeneID}, \mathit{desc}: \textbf{Description}, \\ \mathit{loc}: \textbf{Locus}, \mathit{seq}: \textbf{DNA}, \mathit{mRNA}: \{\textbf{mRNA}\}\rangle$$

$$\text{Protein} = \langle \mathit{id}: \textbf{ProteinID}, \mathit{desc}: \textbf{Description}, \\ \mathit{seq}: \textbf{AminoAcidSeq}, \mathit{pept}: \{\textbf{PeptideSeq}\}\rangle$$

Clearly, Gene $\preceq$ BioSequence and Protein $\preceq$ BioSequence, and thus {Gene} $\preceq$ {BioSequence} and {Protein} $\preceq$ {BioSequence}. Suppose we have an extensive collection of sequence data, about both genes and proteins, that was downloaded from a public database. If we would like to check whether the sequences have changed, we can write a dataflow that accepts a set of values of type BioSequence. $\square$

Defining the subtyping relation based on a partial order of the base types, together with condition (2.1), extends that partial order to complex types, as we prove in Proposition 2.10. We first need the following lemma.

Lemma 2.9. *Let τ and μ be complex types. If $\tau \preceq \mu$ then $[\![\tau]\!] \subseteq [\![\mu]\!]$.*

Proof. Let $\mu \in \mathcal{T}$ such that $\tau \preceq \mu$ holds. We prove $[\![\tau]\!] \subseteq [\![\mu]\!]$ by induction on the structure of τ.

If $\tau = \bot$, then $[\![\tau]\!] = \emptyset \subseteq [\![\mu]\!]$, for any μ as desired.

If τ is a base type, then μ is a base type as well, and $[\![\tau]\!] \subseteq [\![\mu]\!]$ follows from Definition 2.7.

If τ is a set type of the form $\{\tau'\}$, then μ must be a set type of the form $\{\mu'\}$, with $\tau' \preceq \mu'$ (Definition 2.7). To show $[\![\tau]\!] \subseteq [\![\mu]\!]$, let $v \in [\![\tau]\!]$. We have $v \subseteq [\![\tau']\!]$ and, by induction, $[\![\tau']\!] \subseteq [\![\mu']\!]$, and thus $v \in [\![\mu]\!]$ as desired.

If τ is a tuple type of the form $\langle l_1 \colon \tau_1, \ldots, l_n \colon \tau_n \rangle$, than μ must be a tuple type of the form $\langle l_1 \colon \mu_1, \ldots, l_m \colon \mu_m \rangle$ with $m \leq n$, and $\tau_i \preceq \mu_i$ for $i \in \{1, \ldots, m\}$ (Definition 2.7). To show $[\![\tau]\!] \subseteq [\![\mu]\!]$, let $v = \langle l_1 \colon v_1, \ldots, l_q \colon v_q \rangle \in [\![\tau]\!]$, with $q \geq n$. Then, by induction, $v_i \in [\![\tau_i]\!] \subseteq [\![\mu_i]\!]$ for $i \in \{1, \ldots, m\}$, with $q \geq n \geq m$. Thus $v \in [\![\langle l_1 \colon \mu_1, \ldots, l_m \colon \mu_m \rangle]\!] = [\![\mu]\!]$ as desired. □

Proposition 2.10. *The subtyping relation $\preceq$ on $\mathcal{T}$ is a partial order relation.*

Proof. We prove this proposition by induction on the structure of τ. We already know from Definition 2.7 that $\preceq$ is a partial order relation on $\mathcal{B}$. We show it also holds for $\bot$, set types and tuple types. We need to prove the following:

1. $\preceq$ is reflexive: $\forall\, \tau \in \mathcal{T} \colon \tau \preceq \tau$,
2. $\preceq$ is anti-symmetric: $\forall\, \tau, \mu \in \mathcal{T} \colon \tau \preceq \mu \,\wedge\, \mu \preceq \tau \implies \tau = \mu$,
3. $\preceq$ is transitive: $\forall\, \tau, \mu, \nu \in \mathcal{T} \colon \tau \preceq \mu \,\wedge\, \mu \preceq \nu \implies \tau \preceq \nu$.

1. We prove that $\preceq$ is reflexive:

   If $\tau = \bot$, then, by Definition 2.7, $\bot \preceq \nu$ holds for all $\nu \in \mathcal{T}$. Obviously $\bot \preceq \bot$.

   If τ is a set type of the form $\tau = \{\tau'\}$, then, by induction, $\tau' \preceq \tau'$. Therefore, by Definition 2.7, $\{\tau'\} \preceq \{\tau'\}$.

   If τ is a tuple type of the form $\langle l_1 \colon \tau_1, \ldots, l_n \colon \tau_n \rangle$, then, by induction, $\tau_i \preceq \tau_i$ for $i \in \{1, \ldots, n\}$. Thus, by Definition 2.7, $\langle l_1 \colon \tau_1, \ldots, l_n \colon \tau_n \rangle \preceq \langle l_1 \colon \tau_1, \ldots, l_n \colon \tau_n \rangle$.

2. We prove that $\preceq$ is anti-symmetric. Let $\mu \in \mathcal{T}$ such that $\tau \preceq \mu$ and $\mu \preceq \tau$ hold.

   If $\tau = \bot$, then $\mu \preceq \bot$ is only possible if $[\![\mu]\!] = \emptyset$, which, by Definition 2.7, is only possible if $\mu = \bot$. Therefore $\tau = \mu$.

If τ is a set type of the form $\tau = \{\tau'\}$, then, by Definition 2.7, μ must also be a set type of the form $\{\mu'\}$, and $\tau' \preceq \mu'$ and $\mu' \preceq \tau'$ hold. By induction, we have $\tau' = \mu'$, and by Definition 2.7, $\{\tau'\} = \{\mu'\}$, therefore $\tau = \mu$.

If τ is a tuple type of the form $\langle l_1 \colon \tau_1, \ldots, l_n \colon \tau_n \rangle$, then, from $\tau \preceq \mu$ and Definition 2.7, μ must be a tuple type of the form $\langle l_1 \colon \mu_1, \ldots, l_m \colon \mu_m \rangle$, with $m \leq n$ and $\tau_i \preceq \mu_i$ for $i \in \{1, \ldots, m\}$. Since $\mu \preceq \tau$, we know that $n \leq m$ and $\mu_i \preceq \tau_i$ hold as well, for $i \in \{1, \ldots, n\}$. Therefore $m = n$ and, by induction, $\tau_i = \mu_i$ for $i \in \{1, \ldots, m\}$. We have $\tau = \mu$ as desired.

3. We prove that $\preceq$ is transitive. Let $\mu, \nu \in \mathcal{T}$ such that $\tau \preceq \mu$ and $\mu \preceq \nu$ hold.

   If $\tau = \bot$, then, by Definition 2.7, $\bot \preceq \nu$, so trivially $\tau \preceq \nu$.

   If τ is a set type of the form $\tau = \{\tau'\}$, then, by Definition 2.7, μ and ν must be set types of the respective forms $\{\mu'\}$ and $\{\nu'\}$, for which $\tau' \preceq \mu'$ and $\mu' \preceq \nu'$ hold. By induction, $\tau' \preceq \nu'$, and by Definition 2.7, $\{\tau'\} \preceq \{\nu'\}$. We have $\tau \preceq \nu$ as desired.

   If τ is a tuple type of the form $\langle l_1 \colon \tau_1, \ldots, l_n \colon \tau_n \rangle$, then, by Definition 2.7, μ and ν must also be tuple types of the respective forms $\langle l_1 \colon \mu_1, \ldots, l_m \colon \mu_m \rangle$, with $m \leq n$, and $\langle l_1 \colon \nu_1, \ldots, l_q \colon \nu_q \rangle$, with $q \leq m$. Since $q \leq m \leq n$, by induction, $\tau_i \preceq \nu_i$ for $i \in \{1, \ldots, q\}$. By Definition 2.7, $\langle l_1 \colon \tau_1, \ldots, l_n \colon \tau_n \rangle \preceq \langle l_1 \colon \nu_1, \ldots, l_q \colon \nu_q \rangle$, therefore $\tau \preceq \nu$.

□

2.2.2 Compatible types

As illustrated in Example 2.8, there are situations when it is convenient to use one supertype τ for operations on values from different subtypes of τ. In such cases we can call the subtypes of τ compatible. In this section, we formally define *compatible types*, and define their *join*, which is their least general supertype.

Definition 2.11 (Compatible types). For two complex types τ and μ, we define when they are *compatible*. If so, we define their *join*, which is again a complex type, denoted by $\tau \vee \mu$.

- If either of the two types τ and μ is the bottom type, then τ and μ are compatible. Let $\mu = \bot$. We define their join as follows:

$$\tau \vee \bot = \tau.$$

- Two base types τ and μ are compatible if and only if they have a least upper bound w.r.t. $\preceq$. We define $\tau \vee \mu$ to be this least upper bound.
- Two set types $\{\tau\}$ and $\{\mu\}$ are compatible if and only if τ and μ are compatible. We define their join as follows:
$$\{\tau\} \vee \{\mu\} = \{\tau \vee \mu\}.$$
- Two tuple types τ and μ are compatible if and only if they have a non-empty set $\{l_1, \ldots, l_k\}$ of all common distinct labels, such that $\tau = \langle l_1 : \tau_1, \ldots, l_k : \tau_k, \ldots \rangle$ and $\mu = \langle l_1 : \mu_1, \ldots, l_k : \mu_k, \ldots \rangle$, and τ_i and μ_i are compatible for $i \in \{1, \ldots, k\}$. We define their join as follows:
$$\tau \vee \mu = \langle l_1 : \tau_1 \vee \mu_1, \ldots, l_k : \tau_k \vee \mu_k \rangle.$$

Example 2.12. We continue Example 2.8. Types Gene and Protein are compatible, and Gene $\vee$ Protein = BioSequence. Types $\{\textbf{ProteinID}\}$ and $\{\textbf{AminoAcidSeq}\}$ are not compatible, nor are the types

$$\langle \mathit{gene} : \textbf{GeneID}, \mathit{prot} : \{\textbf{ProteinID}\} \rangle$$

and

$$\langle \mathit{gene} : \textbf{NucleotideSeq}, \mathit{prot} : \{\textbf{AminoAcidSeq}\} \rangle .$$

□

Proposition 2.13. *Let τ and μ be two complex types. If $\tau \vee \mu$ exists, then it is the least upper bound of τ and μ w.r.t. $\preceq$, i.e.,*

1. *$\tau \preceq \tau \vee \mu$ and $\mu \preceq \tau \vee \mu$,*
2. *$\forall \nu \in \mathcal{T} : \tau \preceq \nu$ and $\mu \preceq \nu \implies \tau \vee \mu \preceq \nu$.*

Proof. We prove the proposition by induction on the structure of τ. We already know from Definition 2.11 that if τ is a base type and μ is a base type, then $\tau \vee \mu$ is the least upper bound of τ and μ w.r.t. $\preceq$. We show the proposition also holds for $\bot$, set types and tuple types. Note that to prove $\tau \preceq \tau \vee \mu$ and $\mu \preceq \tau \vee \mu$, it is sufficient to show that $\tau \preceq \tau \vee \mu$, because $\tau \vee \mu = \mu \vee \tau$.
Let $\tau = \bot$.

1. By Definition 2.11, $\tau \vee \mu = \bot \vee \mu = \mu$. By Definition 2.7, $\tau = \bot \preceq \mu = \tau \vee \mu$.
2. By Definition 2.11, $\tau \vee \mu = \mu \preceq \nu$.

Let τ be a set type of the form $\{\tau'\}$. Then, by Definition 2.11, μ is also a set type of the form $\{\mu'\}$, and τ' and μ' are compatible. Therefore, $\tau' \vee \mu'$ exists.

1. By induction, $\tau' \preceq \tau' \vee \mu'$. By Definition 2.7, $\{\tau'\} \preceq \{\tau' \vee \mu'\} = \{\tau'\} \vee \{\mu'\}$. Hence $\tau \preceq \tau \vee \mu$.
2. By Definition 2.7, ν must also be a set type of the form $\{\nu'\}$, and both $\tau' \preceq \nu'$ and $\mu' \preceq \nu'$ hold. By induction, $\tau' \vee \mu' \preceq \nu'$ holds. Hence, $\tau \vee \mu = \{\tau' \vee \mu'\} \preceq \{\nu'\} = \nu$.

Let τ be a tuple type. Then, by Definition 2.11, μ is necessarily a tuple type. Moreover, τ and μ have a non-empty set of all common distinct labels, say $\{l_1, \ldots, l_k\}$, such that $\tau = \langle l_1 : \tau_1, \ldots, l_k : \tau_k, \ldots \rangle$, $\mu = \langle l_1 : \mu_1, \ldots, l_k : \mu_k, \ldots \rangle$, and τ_i and μ_i are compatible for $i \in \{1, \ldots, k\}$. Therefore, $\tau_i \vee \mu_i$ exists for $i \in \{1, \ldots, k\}$.

1. For $i \in \{1, \ldots, k\}$, we know by induction that $\tau_i \preceq \tau_i \vee \mu_i$. By Definition 2.7, $\langle l_1 : \tau_1, \ldots, l_k : \tau_k, \ldots \rangle \preceq \langle l_1 : \tau_1 \vee \mu_1, \ldots, l_k : \tau_k \vee \mu_k \rangle$, hence $\tau \preceq \tau \vee \mu$.
2. By Definition 2.7, ν must also be a tuple type. As $\tau \preceq \nu$, ν can be written as $\langle l_1 : \nu_1, \ldots, l_n : \nu_n \rangle$, with $l_{k+1}, \ldots, l_n$ labels common with τ. As $\mu \preceq \nu$, ν can be written as $\langle l_1 : \nu_1, \ldots, l_m : \nu_m \rangle$, with $l_{k+1}, \ldots, l_m$ labels common with μ. As all common labels of τ and μ are in the set $\{l_1, \ldots, l_k\}$, we have $n = m = k$, and thus $\tau_i \leq \nu_i$ and $\mu_i \leq \nu_i$, for $i \in \{1, \ldots, k\}$. By induction, $\tau_i \vee \mu_i \leq \nu_i$, for $i \in \{1, \ldots, k\}$. Hence, $\tau \vee \mu = \langle l_1 : \tau_1 \vee \mu_1, \ldots, l_k : \tau_k \vee \mu_k \rangle \preceq \langle l_1 : \nu_1, \ldots, l_k : \nu_k \rangle = \nu$.

□

2.3 Abstract services

A scientific workflow is often viewed as a computational task that is composed from "simpler" tasks [SGBB01, BF05, FKSS08, DF08]. These "simple" tasks may be calls to various external services, such as the NCBI BLAST webservice; or calls to various library functions, such as addition for numbers, or concatenation for strings, or the application of XQuery to an XML document. Moreover, one dataflow can serve as a task in another dataflow, becoming its subdataflow.

In our model, the composition of a dataflow is structured using the programming constructs of nested relational calculus. To model the diverse tasks, we use *service-call expressions*. In this section, we define the necessary concepts for service-call expressions.

We use abstract *service names* to denote services. The type system requires *signatures* to be attached to these names.

Definition 2.14 (Service names, signatures, signature assignment)**.** Assume a countably infinite set $\mathcal{N}$ of *service names*. We define a *signature* to be an expression of the form $\tau_1 \times \ldots \times \tau_n \to \tau_{out}$, where $\tau_1, \ldots, \tau_n$ and τ_{out} are complex types. We use $\mathcal{S}$ to denote the set of all possible signatures, i.e.,

$$\mathcal{S} \overset{def}{=} \{\tau_1 \times \ldots \times \tau_n \to \tau_{out} \mid n \in \mathbb{N} \wedge (\tau_1, \ldots, \tau_n, \tau_{out}) \in \mathcal{T}^{n+1}\}.$$

For a set N of service names, i.e., $N \subseteq \mathcal{N}$, we define a *signature assignment* over N as a mapping Θ from N to $\mathcal{S}$.

Only at the time of execution of a dataflow we provide a meaning for its service names by assigning them *service functions*, which are, for the time being, merely abstract non-deterministic functions. In other words, we use service functions to model the input-output behaviour of services. We assume service functions to be total[‡]

Definition 2.15 (Service functions, function assignment)**.** We define a *service function* as a relation *IO* from $[\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$ to $[\![\tau_{out}]\!]$, for some complex types $\tau_1, \ldots, \tau_n, \tau_{out}$, that is *total* in the following sense:

$$\forall (w_1, \ldots, w_n) \in [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!].\ \exists v \in [\![\tau_{out}]\!] \colon (w_1, \ldots, w_n, v) \in IO. \quad (2.2)$$

We use $\mathcal{F}$ to denote the set of all possible service functions.

For a set N of service names, i.e., $N \subseteq \mathcal{N}$, we define a *function assignment* over N as a mapping ζ from N to $\mathcal{F}$.

Note that service functions may be non-deterministic — they may have more than one output related to a given input. This is especially important for modelling external services, which we have no control over. For example, the internal database of an on-line service may be updated (e.g., NCBI), or the service may fail from time to time, and produce an error value instead of the expected output value.

Before the execution of a dataflow, we have thus both a signature assignment and a function assignment for its service names. Care must be taken such that this function assignment is *consistent* with the chosen signature assignment.

[‡]Although we have no control over external services, which the abstract services generally represent, we can guarantee totality by using wrapper relations (see Section A.2).

Definition 2.16 (Consistency). Let $N \subseteq \mathcal{N}$. Let Θ be a signature assignment over N, and ζ a function assignment over N. We call ζ *consistent* with Θ, if the following holds:

$$\forall f \in N \colon \Theta(f) = \tau_1 \times \ldots \times \tau_n \to \tau_{out} \implies \zeta(f) \subseteq [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!] \times [\![\tau_{out}]\!] \,.$$

2.4 NRC expressions

In our model, we use nested relational calculus (NRC) to provide the "glue" for connecting various tasks occurring in a dataflow. Nested relational calculus, introduced by Buneman et al. [BNTW95], is a query language over nested collections of tuples. Buneman et al. consider three types of collections, i.e., sets, bags and ordered lists. However, we have chosen to work with sets, because other collection types can be modelled as sets of tuples. For an ordered list, for example, one could use a numerical tuple attribute that indicates the order in the list. Moreover, in our study of use cases, we have noticed that in a significant number of cases, order matters only for visualisation of data.

We naturally augment NRC with service calls, which, for the time being, refer to some abstract services. A dataflow specification containing only abstract services is often referred to as a workflow *template* or an *abstract workflow.* In Section 3.2 we have to be more specific and distinguish between external services and subdataflows. We then provide a mechanism to bind the abstract services to either external services or subdataflows, essentially providing a translation of the abstract specification into a concrete one. Such a concrete dataflow specification is often referred to as a workflow *instance*, or an *executable workflow.*

Formally, Definition 2.17 defines the set of NRC expressions, the grammar of the corresponding abstract syntax is shown in Figure 2.3.

Definition 2.17 (NRC expressions). Assume a countably infinite set $\mathcal{X}$ of variables.[§] We define the set $\mathcal{NRC}$ of *NRC expressions* as the smallest set satisfying the following:

[§]We assume $\mathcal{X}$ and $\mathcal{A}$ to be pairwise disjoint.

$$
\begin{array}{rll}
\mathsf{a} \in \mathcal{A} & \Longrightarrow \mathsf{a} \in \mathcal{NRC} & \text{(constant)} \\
x \in \mathcal{X} & \Longrightarrow x \in \mathcal{NRC} & \text{(variable)} \\
 & \varnothing \in \mathcal{NRC} & \text{(empty set)} \\
e \in \mathcal{NRC} & \Longrightarrow \{e\} \in \mathcal{NRC} & \text{(set)} \\
e_1, e_2 \in \mathcal{NRC} & \Longrightarrow e_1 \cup e_2 \in \mathcal{NRC} & \text{(union)} \\
e \in \mathcal{NRC} & \Longrightarrow \bigcup e \in \mathcal{NRC} & \text{(flatten)} \\
\begin{array}{r} e_1, \ldots, e_n \in \mathcal{NRC} \\ \text{and } \{l_1, \ldots, l_n\} \subseteq \mathcal{L} \end{array} & \Longrightarrow \langle l_1 : e_1, \ldots, l_n : e_n \rangle \in \mathcal{NRC} & \text{(tuple)} \\
e \in \mathcal{NRC} \wedge l \in \mathcal{L} & \Longrightarrow e.l \in \mathcal{NRC} & \text{(projection)} \\
e_1, e_2 \in \mathcal{NRC} \wedge x \in \mathcal{X} & \Longrightarrow \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2 \in \mathcal{NRC} & \text{(for)} \\
e_1, e_2 \in \mathcal{NRC} & \Longrightarrow e_1 = e_2 \in \mathcal{NRC} & \text{(equality test)} \\
e \in \mathcal{NRC} & \Longrightarrow e = \varnothing \in \mathcal{NRC} & \text{(emptiness test)} \\
e_0, e_1, e_2 \in \mathcal{NRC} & \Longrightarrow \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \in \mathcal{NRC} & \text{(if)} \\
e_1, e_2 \in \mathcal{NRC} \wedge x \in \mathcal{X} & \Longrightarrow \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 \in \mathcal{NRC} & \text{(let)} \\
e_1, \ldots, e_n \in \mathcal{NRC} \wedge f \in \mathcal{N} & \Longrightarrow f(e_1, \ldots, e_n) \in \mathcal{NRC} & \text{(service call)}
\end{array}
$$

We require that all labels in a tuple-expression must be distinct.

For convenience, we frequently abbreviate "NRC expression" to "expression". From now on, unless otherwise stated, we use the following notations:

- a is a constant; x is a variable,
- l, l_1, …, l_n are labels,

$$
\begin{array}{rcl}
\mathit{Expr} & \rightarrow & \mathit{BaseExpr} \mid \mathit{CompositeExpr} \\
\mathit{BaseExpr} & \rightarrow & \mathit{Constant} \mid \mathit{Variable} \mid \text{“}\varnothing\text{”} \\
\mathit{CompositeExpr} & \rightarrow & \text{“\{”}\ \mathit{Expr}\ \text{“\}”} \mid \mathit{Expr}\ \text{“}\cup\text{”}\ \mathit{Expr} \mid \text{“}\bigcup\text{”}\ \mathit{Expr} \mid \\
 & & \text{“}\langle\text{”}\ \mathit{Element}\ (\text{“,”}\ \mathit{Element})^*\ \text{“}\rangle\text{”} \mid \mathit{Expr}\ \text{“.”}\ \mathit{Label} \mid \\
 & & \text{“}\mathsf{for}\text{”}\ \mathit{Variable}\ \text{“}\mathsf{in}\text{”}\ \mathit{Expr}\ \text{“}\mathsf{return}\text{”}\ \mathit{Expr} \mid \\
 & & \mathit{Expr}\ \text{“=”}\ \mathit{Expr} \mid \mathit{Expr}\ \text{“}= \varnothing\text{”} \mid \\
 & & \text{“}\mathsf{if}\text{”}\ \mathit{Expr}\ \text{“}\mathsf{then}\text{”}\ \mathit{Expr}\ \text{“}\mathsf{else}\text{”}\ \mathit{Expr} \mid \\
 & & \text{“}\mathsf{let}\text{”}\ \mathit{Variable}\ \text{“:=”}\ \mathit{Expr}\ \text{“}\mathsf{in}\text{”}\ \mathit{Expr} \mid \\
 & & \mathit{ServiceName}\ \text{“(”}\ \mathit{Expr}\ (\text{“,”}\ \mathit{Expr})^*\ \text{“)”} \\
\mathit{Element} & \rightarrow & \mathit{Label}\ \text{“:”}\ \mathit{Expr} \\
\mathit{Constant} & \rightarrow & \mathsf{a} \in \mathcal{A} \\
\mathit{Variable} & \rightarrow & x \in \mathcal{X} \\
\mathit{Label} & \rightarrow & l \in \mathcal{L} \\
\mathit{ServiceName} & \rightarrow & f \in \mathcal{N}
\end{array}
$$

Figure 2.3: Abstract syntax of NRC expressions

- f and g are service names,
- **b** is a base type; τ, μ, τ', τ_1, ..., τ_n are complex types,
- e, e', e_0, ..., e_n are NRC expressions.

Definition 2.18 (Free variables). The *set of free variables* of an expression e, denoted by $FV(e)$, is defined by the following rules. Note that $FV(e) \subseteq \mathcal{X}$.

$$\frac{}{FV(\mathsf{a}) = \emptyset} \qquad \frac{}{FV(x) = \{x\}} \qquad \frac{}{FV(\varnothing) = \emptyset} \qquad \frac{e = \{e'\}}{FV(e) = FV(e')}$$

$$\frac{e = e_1 \cup e_2}{FV(e) = FV(e_1) \cup FV(e_2)} \qquad \frac{e = \bigcup e'}{FV(e) = FV(e')}$$

$$\frac{e = \langle l_1 : e_1, \ldots, l_n : e_n \rangle}{FV(e) = FV(e_1) \cup \cdots \cup FV(e_n)} \qquad \frac{e = e'.l}{FV(e) = FV(e')}$$

$$\frac{e = \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2}{FV(e) = FV(e_1) \cup (FV(e_2) \setminus \{x\})} \qquad \frac{e = (e_1 = e_2)}{FV(e) = FV(e_1) \cup FV(e_2)}$$

$$\frac{e = (e' = \varnothing)}{FV(e) = FV(e')} \qquad \frac{e = \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2}{FV(e) = FV(e_0) \cup FV(e_1) \cup FV(e_2)}$$

$$\frac{e = \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2}{FV(e) = FV(e_1) \cup (FV(e_2) \setminus \{x\})} \qquad \frac{e = f(e_1, \ldots, e_n)}{FV(e) = FV(e_1) \cup \cdots \cup FV(e_n)}$$

Definition 2.19 (Service names). The *set of service names* of an expression e, denoted by $SN(e)$, is defined by the following rules. Note that $SN(e) \subseteq \mathcal{N}$.

$$\frac{}{SN(\mathsf{a}) = \emptyset} \qquad \frac{}{SN(x) = \emptyset} \qquad \frac{}{SN(\varnothing) = \emptyset} \qquad \frac{e = \{e'\}}{SN(e) = SN(e')}$$

$$\frac{e = e_1 \cup e_2}{SN(e) = SN(e_1) \cup SN(e_2)} \qquad \frac{e = \bigcup e'}{SN(e) = SN(e')}$$

$$\frac{e = \langle l_1 : e_1, \ldots, l_n : e_n \rangle}{SN(e) = SN(e_1) \cup \cdots \cup SN(e_n)} \qquad \frac{e = e'.l}{SN(e) = SN(e')}$$

$$\frac{e = \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2}{SN(e) = SN(e_1) \cup SN(e_2)} \qquad \frac{e = (e_1 = e_2)}{SN(e) = SN(e_1) \cup SN(e_2)}$$

$$\frac{e = (e' = \varnothing)}{SN(e) = SN(e')} \qquad \frac{e = \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2}{SN(e) = SN(e_0) \cup SN(e_1) \cup SN(e_2)}$$

$$\frac{e = \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2}{SN(e) = SN(e_1) \cup SN(e_2)} \qquad \frac{e = f(e_1, \ldots, e_n)}{SN(e) = \{f\} \cup SN(e_1) \cup \cdots \cup SN(e_n)}$$

2.4.1 Subexpression path of an NRC expression

Intuitively, a *subexpression* of an expression e is an expression occurring in e, possibly e itself. Similarly to a subvalue of a complex value, a subexpression of an expression e may have various occurrences in e. Again, we need a formal way to distinguish these occurrences. Thereto, we introduce the notion of a *subexpression path.*

A subexpression path of an expression e is a sequence over $\mathcal{NRC} \cup \mathcal{L} \cup \mathbb{N}$, where $\mathbb{N}$ is the set of natural numbers.[¶] A subexpression path of e starts with e itself and consists of subexpressions of e, possibly preceded by labels from $\mathcal{L}$ or natural numbers.

Formally, we define a subexpression path of an expression by the rules given in Definition 2.20. Note that expressions a, x and $\varnothing$ are covered by the first rule, i.e., they have no other subexpressions except themselves. We use symbol "Φ" to denote a subexpression path, and notation $\Phi \leftarrow\!\circ\; e$ for "Φ is a subexpression path of e". When a subexpression path Φ of e is not $[e]$ itself, we call it a *proper subexpression path* of e, and use $\Phi \leftarrow\!\dot{\circ}\; e$ as notation.

Definition 2.20 (Subexpression path of an NRC expression).

$$\frac{e \in \mathcal{NRC}}{[e] \leftarrow\!\circ\; e}$$

$$\frac{e = \{e'\} \;\text{ or }\; e = \bigcup e' \;\text{ or }\; e = e'.l \;\text{ or }\; e = (e' = \varnothing) \qquad \Phi \leftarrow\!\circ\; e'}{[e] \cdot \Phi \leftarrow\!\circ\; e}$$

$$\frac{\begin{array}{c} e = e_1 \cup e_2 \;\text{ or }\; e = (e_1 = e_2) \;\text{ or }\; e = \mathsf{for}\; x \;\mathsf{in}\; e_1 \;\mathsf{return}\; e_2 \\ \text{or }\; e = \mathsf{let}\; x := e_1 \;\mathsf{in}\; e_2 \qquad \Phi \leftarrow\!\circ\; e_i \qquad i \in \{1,2\} \end{array}}{[e, i] \cdot \Phi \leftarrow\!\circ\; e}$$

$$\frac{e = \mathsf{if}\; e_0 \;\mathsf{then}\; e_1 \;\mathsf{else}\; e_2 \qquad \Phi \leftarrow\!\circ\; e_i \qquad i \in \{0,1,2\}}{[e, i] \cdot \Phi \leftarrow\!\circ\; e}$$

$$\frac{e = \langle l_1 : e_1, \ldots, l_n : e_n \rangle \qquad \Phi \leftarrow\!\circ\; e_i \qquad i \in \{1,\ldots,n\}}{[e, l_i] \cdot \Phi \leftarrow\!\circ\; e}$$

$$\frac{e = f(e_1, \ldots, e_n) \qquad \Phi \leftarrow\!\circ\; e_i \qquad i \in \{1,\ldots,n\}}{[e, i] \cdot \Phi \leftarrow\!\circ\; e}$$

[¶]We assume $\mathcal{NRC}$, $\mathcal{L}$, and $\mathbb{N}$ to be pairwise disjoint.

We use subexpression paths to define runs of NRC expressions in Section 3.1.1, and subvalue provenance in Section 4.2. In this context, we do not need a subexpression path to the bound variable of a for-expression, nor to the bound variable of a let-expression.

Example 2.21. Let e be the following expression:

```
for $x$ in $\{x\}$ return
    if $x = \mathsf{a}$ then $f(x)$ else $g(x)$,
```

then e has the following subexpression paths.

$$\begin{array}{l}
[e] \\
[e, 1, \{x\}] \\
[e, 1, \{x\}, x] \\
[e, 2, \mathsf{if}\ x = \mathsf{a}\ \mathsf{then}\ f(x)\ \mathsf{else}\ g(x)] \\
[e, 2, \mathsf{if}\ x = \mathsf{a}\ \mathsf{then}\ f(x)\ \mathsf{else}\ g(x), 0, x = \mathsf{a}] \\
[e, 2, \mathsf{if}\ x = \mathsf{a}\ \mathsf{then}\ f(x)\ \mathsf{else}\ g(x), 0, x = \mathsf{a}, 1, x] \\
[e, 2, \mathsf{if}\ x = \mathsf{a}\ \mathsf{then}\ f(x)\ \mathsf{else}\ g(x), 0, x = \mathsf{a}, 2, \mathsf{a}] \\
[e, 2, \mathsf{if}\ x = \mathsf{a}\ \mathsf{then}\ f(x)\ \mathsf{else}\ g(x), 1, f(x)] \\
[e, 2, \mathsf{if}\ x = \mathsf{a}\ \mathsf{then}\ f(x)\ \mathsf{else}\ g(x), 1, f(x), 1, x] \\
[e, 2, \mathsf{if}\ x = \mathsf{a}\ \mathsf{then}\ f(x)\ \mathsf{else}\ g(x), 2, g(x)] \\
[e, 2, \mathsf{if}\ x = \mathsf{a}\ \mathsf{then}\ f(x)\ \mathsf{else}\ g(x), 2, g(x), 1, x]
\end{array}$$

Note that variable x is a subexpression of e, but there are actually four different occurrences of x in e, according to Definition 2.20. □

2.4.2 Well-typedness of an NRC expression

After writing an expression, we need to (i) assign input types to the free variables, and (ii) assign signatures to the service names used in the expression. The latter is accomplished by a signature assignment (Definition 2.14). The former is accomplished by a *type assignment*, which we subsequently define.

Definition 2.22 (Type assignment)**.** Let $X \subseteq \mathcal{X}$. We define a *type assignment* over X as a mapping Γ from X to the set of complex types. We use $\mathcal{TA}$ to denote the set of all possible type assignments.

Next we define an operation on type assignments.

Definition 2.23. Let $X \subseteq \mathcal{X}$. Let Γ be a type assignment over X, $x \in \mathcal{X}$, and τ a complex type. We define $\Gamma[x \mapsto \tau]$ as the type assignment over $X \cup \{x\}$ that is equal to Γ, except for the variable x that is assigned the type τ.

We are ready to formally define a type system for NRC expressions. Since this system is known from the literature [BNTW95], we omit informal explanation.

Let e be an NRC expression, Γ a type assignment over $FV(e)$, and Θ a signature assignment over $SN(e)$.

Rules T.1–T.15, presented in Definition 2.24, define whether *e is well-typed under Γ and Θ*, and if so, determine possible types for e. The rules infer judgements of the form $\Gamma, \Theta \vdash e\colon \tau$, meaning "$\tau$ is a possible type inferred for e under Γ and Θ". When there is no type τ, such that $\Gamma, \Theta \vdash e\colon \tau$ can be derived by these rules, we say that e is not well-typed under Γ and Θ.

Definition 2.24 (Well-typedness of an expression).

$$\frac{\mathsf{a} \in [\![\mathbf{b}]\!]}{\Gamma, \Theta \vdash \mathsf{a}\colon \mathbf{b}}\ \text{T.1} \qquad \frac{}{\Gamma, \Theta \vdash x\colon \Gamma(x)}\ \text{T.2} \qquad \frac{}{\Gamma, \Theta \vdash \varnothing\colon \{\bot\}}\ \text{T.3}$$

$$\frac{\Gamma, \Theta \vdash e\colon \mu \qquad \mu \preceq \tau}{\Gamma, \Theta \vdash e\colon \tau}\ \text{T.4} \qquad \frac{\Gamma, \Theta \vdash e\colon \tau}{\Gamma, \Theta \vdash \{e\}\colon \{\tau\}}\ \text{T.5}$$

$$\frac{\Gamma, \Theta \vdash e_1\colon \{\tau\} \qquad \Gamma, \Theta \vdash e_2\colon \{\tau\}}{\Gamma, \Theta \vdash e_1 \cup e_2\colon \{\tau\}}\ \text{T.6} \qquad \frac{\Gamma, \Theta \vdash e\colon \{\{\tau\}\}}{\Gamma, \Theta \vdash \bigcup e\colon \{\tau\}}\ \text{T.7}$$

$$\frac{\forall i \in \{1, \ldots, n\}\colon \Gamma, \Theta \vdash e_i\colon \tau_i}{\Gamma, \Theta \vdash \langle l_1\colon e_1, \ldots, l_n\colon e_n\rangle\colon \langle l_1\colon \tau_1, \ldots, l_n\colon \tau_n\rangle}\ \text{T.8}$$

$$\frac{\Gamma, \Theta \vdash e\colon \langle \ldots, l\colon \tau, \ldots\rangle}{\Gamma, \Theta \vdash e.l\colon \tau}\ \text{T.9}$$

$$\frac{\Gamma, \Theta \vdash e_1\colon \{\tau_1\} \qquad \Gamma[x \mapsto \tau_1], \Theta \vdash e_2\colon \tau_2}{\Gamma, \Theta \vdash \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2\colon \{\tau_2\}}\ \text{T.10}$$

$$\frac{\Gamma, \Theta \vdash e_1\colon \tau \qquad \Gamma, \Theta \vdash e_2\colon \tau}{\Gamma, \Theta \vdash e_1 = e_2\colon \mathbf{Boolean}}\ \text{T.11} \qquad \frac{\Gamma, \Theta \vdash e\colon \{\tau\}}{\Gamma, \Theta \vdash e = \varnothing\colon \mathbf{Boolean}}\ \text{T.12}$$

$$\frac{\Gamma, \Theta \vdash e_0\colon \mathbf{Boolean} \qquad \Gamma, \Theta \vdash e_1\colon \tau \qquad \Gamma, \Theta \vdash e_2\colon \tau}{\Gamma, \Theta \vdash \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\colon \tau}\ \text{T.13}$$

$$\frac{\Gamma, \Theta \vdash e_1\colon \tau_1 \qquad \Gamma[x \mapsto \tau_1], \Theta \vdash e_2\colon \tau_2}{\Gamma, \Theta \vdash \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2\colon \tau_2}\ \text{T.14}$$

$$\frac{\forall i \in \{1, \ldots, n\}\colon \Gamma, \Theta \vdash e_i\colon \tau_i \qquad \Theta(f) = \tau_1 \times \cdots \times \tau_n \to \tau_{out}}{\Gamma, \Theta \vdash f(e_1, \ldots, e_n)\colon \tau_{out}}\ \text{T.15}$$

Remark that a constant expression a may have more than one possible type under a given type assignment. Every base type **b**, such that $\mathtt{a} \in [\![\mathbf{b}]\!]$, is a possible type for a. For example, Jane can have both **FirstName** and **String** as its type.

The type system may seem too restrictive, in the following respects:

1. an empty-set expression is only given the explicit type $\{\perp\}$;

2. an equality-test, a union, and an if-expressions are only well-typed if e_1 and e_2 can be given exactly the same type.

Fortunately, these restrictions are alleviated by Rule T.4, that infers $e\colon \tau$ from $e\colon \mu$, provided that μ is a subtype of τ. Indeed,

1. since $\perp \preceq \tau$ for any type τ, we infer that $\varnothing\colon \{\tau\}$, for any type τ;

2. likewise, an equality-test, a union, and an if-expressions are well-typed if e_1 and e_2 can be given compatible types. Indeed, if $e_1\colon \tau_1$ and $e_2\colon \tau_2$ and $\tau_1 \vee \tau_2$ exists, then, by Proposition 2.13, we have $e_1\colon \tau_1 \vee \tau_2$ and $e_2\colon \tau_1 \vee \tau_2$, and thus the expression is well-typed.

2.4.3 Value semantics of an NRC expression

When we want to run an expression, we need to (i) assign input values to the free variables, and (ii) assign service functions to the service names used in the expression. The latter is accomplished by a function assignment (Definition 2.15). The former is accomplished by a *value assignment*, which we subsequently define.

Definition 2.25 (Value assignment). Let $X \subseteq \mathcal{X}$. We define a *value assignment* over X as a sequence σ of pairs of the form (x, v), with $x \in X$ and v a complex value. Every $x \in X$ must appear at least once in σ, but may appear several times.

We define a value assignment as a sequence to allow for reuse of variables. Next we define two operations on value assignments.

Definition 2.26. Let $X \subseteq \mathcal{X}$. Let σ be a value assignment over X, and $x \in X$. We define $get(\sigma, x)$ as the value v, such that the pair (x, v) is the *last* pair in σ having x as its first element.

Definition 2.27. Let $X \subseteq \mathcal{X}$. Let σ be a value assignment over X, $x \in \mathcal{X}$, and v a complex value. We define $add(\sigma, x, v)$ as the sequence obtained from σ by appending the pair (x, v) at the *end* of σ. Note that $add(\sigma, x, v)$ is a value assignment over $X \cup \{x\}$.

Definition 2.28 (Consistency). Let $X \subseteq \mathcal{X}$. If Γ is a type assignment over X, and σ is a value assignment over X, then we say that σ is *consistent* with Γ if, for every $x \in X$, value $get(\sigma, x)$ is a complex value of type $\Gamma(x)$.

We are ready to formally define a system of rules for evaluating NRC expressions. Since these inference rules are known from the literature [BNTW95], we omit informal explanation.

Let e be an NRC expression, σ a value assignment over $FV(e)$, and ζ a function assignment over $SN(e)$.

Rules V.1–V.17, presented in Definition 2.29, define *a result value of* e *under* σ *and* ζ. The rules infer judgements of the form $\sigma, \zeta \models e \Rightarrow v$, meaning "value v is a possible final result of evaluating e under σ and ζ". Recall that there can be more than one possible final result value, if non-deterministic service functions are used in the evaluation. Note that in Rule V.9, some values v_i, $i \in \{1, \ldots, n\}$, may be equal.

Definition 2.29 (Value semantics of an NRC expression).

$$\frac{\mathsf{a} \in \mathcal{A}}{\sigma, \zeta \models \mathsf{a} \Rightarrow \mathsf{a}}\ \text{V.1} \qquad \frac{x \in \mathcal{X}}{\sigma, \zeta \models x \Rightarrow get(\sigma, x)}\ \text{V.2} \qquad \frac{}{\sigma, \zeta \models \varnothing \Rightarrow \emptyset}\ \text{V.3}$$

$$\frac{\sigma, \zeta \models e \Rightarrow v}{\sigma, \zeta \models \{e\} \Rightarrow \{v\}}\ \text{V.4}$$

$$\frac{\sigma, \zeta \models e_1 \Rightarrow v_1 \qquad \sigma, \zeta \models e_2 \Rightarrow v_2 \qquad v_1 \text{ and } v_2 \text{ are sets}}{\sigma, \zeta \models e_1 \cup e_2 \Rightarrow v_1 \cup v_2}\ \text{V.5}$$

$$\frac{\sigma, \zeta \models e \Rightarrow \{v_1, \ldots, v_n\} \qquad v_i \text{ is a set, for } i \in \{1, \ldots, n\}}{\sigma, \zeta \models \bigcup e \Rightarrow v_1 \cup \cdots \cup v_n}\ \text{V.6}$$

$$\frac{\forall i \in \{1, \ldots, n\} \colon \sigma, \zeta \models e_i \Rightarrow v_i}{\sigma, \zeta \models \langle l_1 \colon e_1, \ldots, l_n \colon e_n \rangle \Rightarrow \langle l_1 \colon v_1, \ldots, l_n \colon v_n \rangle}\ \text{V.7}$$

$$\frac{\sigma, \zeta \models e \Rightarrow \langle \ldots, l \colon v, \ldots \rangle}{\sigma, \zeta \models e.l \Rightarrow v}\ \text{V.8}$$

$$\frac{\begin{array}{c}\sigma,\zeta \models e_1 \Rightarrow \{w_1,\ldots,w_n\} \text{ with all distinct } w_i \\ \forall i \in \{1,\ldots,n\}\colon add(\sigma,x,w_i),\zeta \models e_2 \Rightarrow v_i\end{array}}{\sigma,\zeta \models \textsf{for } x \textsf{ in } e_1 \textsf{ return } e_2 \Rightarrow \{v_1,\ldots,v_n\}}\ \text{V.9}$$

$$\frac{\sigma,\zeta \models e_1 \Rightarrow v_1 \qquad \sigma,\zeta \models e_2 \Rightarrow v_2 \qquad v_1 = v_2}{\sigma,\zeta \models e_1 = e_2 \Rightarrow \texttt{true}}\ \text{V.10}$$

$$\frac{\sigma,\zeta \models e_1 \Rightarrow v_1 \qquad \sigma,\zeta \models e_2 \Rightarrow v_2 \qquad v_1 \neq v_2}{\sigma,\zeta \models e_1 = e_2 \Rightarrow \texttt{false}}\ \text{V.11}$$

$$\frac{\sigma,\zeta \models e \Rightarrow v \qquad v = \emptyset}{\sigma,\zeta \models e = \varnothing \Rightarrow \texttt{true}}\ \text{V.12} \qquad \frac{\sigma,\zeta \models e \Rightarrow v \qquad v \neq \emptyset}{\sigma,\zeta \models e = \varnothing \Rightarrow \texttt{false}}\ \text{V.13}$$

$$\frac{\sigma,\zeta \models e_0 \Rightarrow \texttt{true} \qquad \sigma,\zeta \models e_1 \Rightarrow v}{\sigma,\zeta \models \textsf{if } e_0 \textsf{ then } e_1 \textsf{ else } e_2 \Rightarrow v}\ \text{V.14}$$

$$\frac{\sigma,\zeta \models e_0 \Rightarrow \texttt{false} \qquad \sigma,\zeta \models e_2 \Rightarrow v}{\sigma,\zeta \models \textsf{if } e_0 \textsf{ then } e_1 \textsf{ else } e_2 \Rightarrow v}\ \text{V.15}$$

$$\frac{\sigma,\zeta \models e_1 \Rightarrow w \qquad add(\sigma,x,w),\zeta \models e_2 \Rightarrow v}{\sigma,\zeta \models \textsf{let } x := e_1 \textsf{ in } e_2 \Rightarrow v}\ \text{V.16}$$

$$\frac{\forall i \in \{1,\ldots,n\}\colon \sigma,\zeta \models e_i \Rightarrow v_i \qquad (v_1,\ldots,v_n,w) \in \zeta(f)}{\sigma,\zeta \models f(e_1,\ldots,e_n) \Rightarrow w}\ \text{V.17}$$

Theorem 2.30 (Soundness). *Let Γ be a type assignment over $FV(e)$, and Θ a signature assignment over $SN(e)$. If σ is consistent with Γ, and ζ is consistent with Θ, and $\Gamma, \Theta \vdash e : \tau$, then the following holds:*

(a) there is a complex value v, such that $\sigma,\zeta \models e \Rightarrow v$;

(b) for all $v \in \mathcal{V}$, if $\sigma,\zeta \models e \Rightarrow v$ then $v \in [\![\tau]\!]$.

Proof. We prove the theorem by induction on the number of rules necessary to derive $\Gamma, \Theta \vdash e : \tau$.

The base case involves Rules T.1–T.3.

For Rule T.1, $e = \mathsf{a}$ and $\tau = \mathbf{b}$.

(a) Clearly, $\mathsf{a} \in \mathcal{A}$. We apply Rule V.1, thus $v = \mathsf{a}$.

(b) Let $v \in \mathcal{V}$, such that $\sigma,\zeta \models \mathsf{a} \Rightarrow v$. The only rule that could have been applied is Rule V.1, so $v = \mathsf{a}$. From Rule T.1, $v \in [\![\mathbf{b}]\!]$.

For Rule T.2, $e = x$ and $\tau = \Gamma(x)$.

(a) We apply Rule V.2, hence $v = get(\sigma, x)$.

(b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models x \Rightarrow v$. The only rule that could have been applied is Rule V.2, so $v = get(\sigma, x)$. We know that σ is consistent with Γ, thus $v \in [\![\Gamma(x)]\!]$.

For Rule T.3, $e = \varnothing$ and $\tau = \{\perp\}$

(a) We apply Rule V.3, thus $v = \emptyset$.

(b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models \varnothing \Rightarrow v$. The only rule that could have been applied is Rule V.3, so $v = \emptyset$. By Definition 2.6, $v \in [\![\{\perp\}]\!]$.

Suppose the theorem holds for all $\Gamma, \Theta \vdash e \colon \tau$ that can be derived by n rules. We now prove it also holds for $\Gamma, \Theta \vdash e \colon \tau$ that can be derived by $n + 1$ rules.

1. Suppose the last rule used to derive $\Gamma, \Theta \vdash e \colon \tau$ was Rule T.4. Then we can apply the induction hypothesis to $\Gamma, \Theta \vdash e \colon \mu$, with $\mu \preceq \tau$, so

   (i) $\exists v' \in \mathcal{V} \colon \sigma, \zeta \models e \Rightarrow v'$;

   (ii) for all $v' \in \mathcal{V}$, if $\sigma, \zeta \models e \Rightarrow v'$ then $v' \in [\![\mu]\!]$.

   (a) From case 1i, simply take $v = v'$.

   (b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models e \Rightarrow v$. From case 1ii, $v \in [\![\mu]\!]$. Since $\mu \preceq \tau$, by Lemma 2.9, $[\![\mu]\!] \subseteq [\![\tau]\!]$. Hence $v \in [\![\tau]\!]$.

2. Suppose the last rule used to derive $\Gamma, \Theta \vdash e \colon \tau$ was Rule T.5. Then $e = \{e'\}$, $\tau = \{\tau'\}$, and we can apply the induction hypothesis to $\Gamma, \Theta \vdash e' : \tau'$, so

   (i) $\exists v' \in \mathcal{V} \colon \sigma, \zeta \models e' \Rightarrow v'$;

   (ii) for all $v' \in \mathcal{V}$, if $\sigma, \zeta \models e' \Rightarrow v'$ then $v' \in [\![\tau']\!]$.

   (a) Take v' from case 2i. We apply Rule V.4, thus $v = \{v'\}$.

   (b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models \{e'\} \Rightarrow v$. The only rule that could have been last applied is Rule V.4, so there is a value v'', such that v'' was produced by evaluation of e' under σ and ζ, and $v = \{v''\}$. From case 2ii, $v'' \in [\![\tau']\!]$. By Definition 2.6, $v = \{v''\} \in [\![\{\tau'\}]\!]$.

3. Suppose the last rule used to derive $\Gamma, \Theta \vdash e : \tau$ was Rule T.6. Then $e = e_1 \cup e_2$, and we can apply the induction hypothesis to $\Gamma, \Theta \vdash e_i : \tau$, for $i \in \{1, 2\}$, so

   (i) $\exists\, v_i \in \mathcal{V} \colon \sigma, \zeta \models e_i \Rightarrow v_i$;

   (ii) for all $v_i \in \mathcal{V}$, if $\sigma, \zeta \models e_i \Rightarrow v_i$ then $v_i \in [\![\tau]\!]$;

   (a) Take v_1 and v_2 from case 3i. We know from case 3ii that v_1 and v_2 are sets. Hence, we apply Rule V.5, thus $v = v_1 \cup v_2$.

   (b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models e_1 \cup e_2 \Rightarrow v$. The only rule that could have been last applied is Rule V.5, so, for $i \in \{1, 2\}$, there is a value v_i', such that v_i' was produced by evaluation of e_i under σ and ζ. Thus $v = v_1' \cup v_2'$. From case 3ii, $v_i' \in [\![\tau]\!]$. By Definition 2.6, $v = v_1' \cup v_2' \in [\![\tau]\!]$.

4. Suppose the last rule used to derive $\Gamma, \Theta \vdash e : \tau$ was Rule T.7. Then $e = \bigcup e'$, $\tau = \{\tau'\}$, and we can apply the induction hypothesis to $\Gamma, \Theta \vdash e' : \{\{\tau'\}\}$, so

   (i) $\exists\, v' \in \mathcal{V} \colon \sigma, \zeta \models e' \Rightarrow v'$;

   (ii) for all $v' \in \mathcal{V}$, if $\sigma, \zeta \models e' \Rightarrow v'$ then $v' \in [\![\{\{\tau'\}\}]\!]$.

   (a) Take v' from case 4i. We know from case 4ii that v' is a set of sets. Hence, we apply Rule V.6, thus $v = \bigcup v'$.

   (b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models \bigcup e' \Rightarrow v$. The only rule that could have been last applied is Rule V.6, so there is a value v'', such that v'' was produced by evaluation of e' under σ and ζ, and $v = \bigcup v''$. From case 4ii, $v'' \in [\![\{\{\tau'\}\}]\!]$. By Definition 2.6, $v = \bigcup v'' \in [\![\{\tau'\}]\!]$.

5. Suppose the last rule used to derive $\Gamma, \Theta \vdash e : \tau$ was Rule T.8. Then $e = \langle l_1 \colon e_1, \ldots, l_n \colon e_n \rangle$, and $\tau = \langle l_1 \colon \tau_1, \ldots, l_n \colon \tau_n \rangle$. We can apply the induction hypothesis to $\Gamma, \Theta \vdash e_i : \tau_i$, for $i \in \{1, \ldots, n\}$, so

   (i) $\exists\, v_i \in \mathcal{V} \colon \sigma, \zeta \models e_i \Rightarrow v_i$;

   (ii) for all $v_i \in \mathcal{V}$, if $\sigma, \zeta \models e_i \Rightarrow v_i$ then $v_i \in [\![\tau_i]\!]$.

   (a) For $i \in \{1, \ldots, n\}$, take v_i from case 5i. We apply Rule V.7, thus $v = \langle l_1 \colon v_1, \ldots, l_n \colon v_n \rangle$.

   (b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models \langle l_1 \colon e_1, \ldots, l_n \colon e_n \rangle \Rightarrow v$. The only rule that could have been last applied is Rule V.7, so, for $i \in \{1, \ldots, n\}$, there is a value v_i', such that v_i' was produced by evaluation of e_i

under σ and ζ. Thus $v = \langle l_1 : v'_1, \ldots, l_n : v'_n \rangle$. From case 5ii, $v'_i \in [\![\tau_i]\!]$. By Definition 2.6, $v = \langle l_1 : v'_1, \ldots, l_n : v'_n \rangle \in [\![\langle l_1 : \tau_1, \ldots, l_n : \tau_n \rangle]\!]$.

6. Suppose the last rule used to derive $\Gamma, \Theta \vdash e : \tau$ was Rule T.9. Then $e = e'.l$, and we can apply the induction hypothesis to $\Gamma, \Theta \vdash e' : \langle \ldots, l \colon \tau, \ldots \rangle$, so

   (i) $\exists v' \in \mathcal{V} \colon \sigma, \zeta \models e' \Rightarrow v'$;

   (ii) for all $v' \in \mathcal{V}$, if $\sigma, \zeta \models e' \Rightarrow v'$ then $v' \in [\![\langle \ldots, l \colon \tau, \ldots \rangle]\!]$.

   (a) Take v' from case 6i. We know from case 6ii that v' is a tuple type with a component labelled by l, so $v' = \langle \ldots, l \colon w, \ldots \rangle$, for some $w \in \mathcal{V}$. We apply Rule V.8, thus $v = w$.

   (b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models e'.l \Rightarrow v$. The only rule that could have been last applied is Rule V.8, so there is a value $\langle \ldots, l \colon v'', \ldots \rangle$, such that it was produced by evaluation of e' under σ and ζ, and $v = v''$. From case 6ii, $\langle \ldots, l \colon v'', \ldots \rangle \in [\![\langle \ldots, l \colon \tau, \ldots \rangle]\!]$. By Definition 2.6, $v = v'' \in [\![\tau]\!]$.

7. Suppose the last rule used to derive $\Gamma, \Theta \vdash e : \tau$ was Rule T.10. Then $e = \textsf{for } x \textsf{ in } e_1 \textsf{ return } e_2$ and $\tau = \{\tau_2\}$. We can apply the induction hypothesis to $\Gamma, \Theta \vdash e_1 : \{\tau_1\}$, so

   (i) $\exists v_1 \in \mathcal{V} \colon \sigma, \zeta \models e_1 \Rightarrow v_1$;

   (ii) for all $v_1 \in \mathcal{V}$, if $\sigma, \zeta \models e_1 \Rightarrow v_1$ then $v_1 \in [\![\{\tau_1\}]\!]$.

   We can also apply the induction hypothesis to $\Gamma[x \mapsto \tau_1] \vdash e_2 : \tau_2$, for all value assignments σ' that are consistent with $\Gamma[x \mapsto \tau_1]$, so

   (iii) $\exists v_2 \in \mathcal{V} \colon \sigma' \models e_2 \Rightarrow v_2$;

   (iv) for all $v_2 \in \mathcal{V}$, if $\sigma' \models e_2 \Rightarrow v_2$ then $v_2 \in [\![\tau_2]\!]$.

   (a) Take v_1 from case 7i. From case 7ii, $v_1 \in [\![\{\tau_1\}]\!]$, so, for $w \in v_1$, value assignments $add(\sigma, x, w)$ are consistent with $\Gamma[x \mapsto \tau_1]$. From case 7iii, there exists a value v_w for each $add(\sigma, x, w)$. We apply Rule V.9, so $v = \{v_w \mid w \in v_1\}$.

   (b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models \textsf{for } x \textsf{ in } e_1 \textsf{ return } e_2 \Rightarrow v$. The only rule that could have been last applied is Rule V.9, so there is a set w that was produced by evaluation of e_1 under σ and ζ; and for $u \in w$, the value assignment $add(\sigma, x, u)$, such that v_u was produced by evaluation of e_2 under $add(\sigma, x, u)$ and ζ. Thus $v = \{v_u \mid u \in w\}$.

From case 7ii, $w \in [\![\{\tau_1\}]\!]$, so each $u \in [\![\tau_1]\!]$, and each $add(\sigma, x, u)$ is consistent with $\Gamma[x \mapsto \tau_1]$. From case 7iv, each $v_u \in [\![\tau_2]\!]$. By Definition 2.6, $v = \{v_u \mid u \in w\} \in [\![\{\tau_2\}]\!]$.

8. Suppose the last rule used to derive $\Gamma, \Theta \vdash e : \tau$ was Rule T.11. Then $e = (e_1 = e_2)$, $\tau =$ **Boolean**, and we can apply the induction hypothesis to $\Gamma, \Theta \vdash e_i : \tau'$, for $i \in \{1, 2\}$, so
   (i) $\exists v_i \in \mathcal{V} \colon \sigma, \zeta \models e_i \Rightarrow v_i$;
   (ii) for all $v_i \in \mathcal{V}$, if $\sigma, \zeta \models e_i \Rightarrow v_i$ then $v_i \in [\![\tau']\!]$.

   (a) Take v_1 and v_2 from case 8i. We know from case 8ii that v_1 and v_2 are elements of $[\![\tau']\!]$. If $v_1 = v_2$, we can apply Rule V.10, so $v = \texttt{true}$. If $v_1 \neq v_2$, we can apply Rule V.11, so $v = \texttt{false}$.
   (b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models e_1 = e_2 \Rightarrow v$. The only rules that could have been last applied are Rule V.10 and Rule V.11. If it was Rule V.10, then $v = \texttt{true} \in [\![\textbf{Boolean}]\!]$. If it was Rule V.11, then $v = \texttt{false} \in [\![\textbf{Boolean}]\!]$.

9. Suppose the last rule used to derive $\Gamma, \Theta \vdash e : \tau$ was Rule T.12. Then $e = (e' = \varnothing)$, $\tau =$ **Boolean**, and we can apply the induction hypothesis to $\Gamma, \Theta \vdash e' : \{\tau'\}$, so
   (i) $\exists v' \in \mathcal{V} \colon \sigma, \zeta \models e' \Rightarrow v'$;
   (ii) for all $v' \in \mathcal{V}$, if $\sigma, \zeta \models e' \Rightarrow v'$ then $v' \in [\![\{\tau'\}]\!]$.

   (a) Take v' from case 9i. We know from case 9ii that v' is a set. If $v' = \emptyset$, we can apply Rule V.12, so $v = \texttt{true}$. If $v' \neq \emptyset$, we can apply Rule V.13, so $v = \texttt{false}$.
   (b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models e' = \varnothing \Rightarrow v$. The only rules that could have been last applied are Rule V.12 and Rule V.13. If it was Rule V.12, then $v = \texttt{true} \in [\![\textbf{Boolean}]\!]$. If it was Rule V.13, then $v = \texttt{false} \in [\![\textbf{Boolean}]\!]$.

10. Suppose the last rule used to derive $\Gamma, \Theta \vdash e : \tau$ was Rule T.13. Then $e = \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, and we can apply the induction hypothesis to $\Gamma, \Theta \vdash e_i : \tau$, so
    (i) for $i \in \{0, 1, 2\}$, $\exists v_i \in \mathcal{V} \colon \sigma, \zeta \models e_i \Rightarrow v_i$;
    (ii) for all $v_0 \in \mathcal{V}$, if $\sigma, \zeta \models e_0 \Rightarrow v_0$ then $v_0 \in [\![\textbf{Boolean}]\!]$;
    (iii) for $i \in \{1, 2\}$, for all $v_i \in \mathcal{V}$, if $\sigma, \zeta \models e_i \Rightarrow v_i$ then $v_i \in [\![\tau]\!]$.

(a) For $i \in \{0,1,2\}$, take v_i from case 10i. We know from case 10ii that $v_0 \in [\![\mathbf{Boolean}]\!]$. If $v_0 = \mathtt{true}$, we can apply Rule V.14, so $v = v_1$. If $v_0 = \mathtt{false}$, we can apply Rule V.15, so $v = v_2$.

(b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Rightarrow v$. The only rules that could have been last applied are Rule V.14 and Rule V.15. If it was Rule V.14, then v was produced by evaluation of e_1 under σ and ζ. If it was Rule V.15, then v was produced by evaluation of e_2 under σ and ζ. From case 10iii, $v \in [\![\tau]\!]$.

11. Suppose the last rule used to derive $\Gamma, \Theta \vdash e : \tau$ was Rule T.14. Then $e = \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2$. We can apply the induction hypothesis to $\Gamma, \Theta \vdash e_1 : \tau'$, so

    (i) $\exists\, v_1 \in \mathcal{V} \colon \sigma, \zeta \models e_1 \Rightarrow v_1$;

    (ii) for all $v_1 \in \mathcal{V}$, if $\sigma, \zeta \models e_1 \Rightarrow v_1$ then $v_1 \in [\![\tau']\!]$.

    We can also apply the induction hypothesis to $\Gamma[x \mapsto \tau'] \vdash e_2 : \tau$, for all value assignments σ' that are consistent with $\Gamma[x \mapsto \tau']$, so

    (iii) $\exists\, v_2 \in \mathcal{V} \colon \sigma' \models e_2 \Rightarrow v_2$;

    (iv) for all $v_2 \in \mathcal{V}$, if $\sigma' \models e_2 \Rightarrow v_2$ then $v_2 \in [\![\tau]\!]$.

    (a) Take v_1 from case 11i. From case 11ii, $v_1 \in [\![\tau']\!]$, so $add(\sigma, x, v_1)$ is consistent with $\Gamma[x \mapsto \tau']$. From case 11iii with $\sigma' = add(\sigma, x, v_1)$, take value v_2. We apply Rule V.16, so $v = v_2$.

    (b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 \Rightarrow v$. The only rule that could have been last applied is Rule V.16, so there is a value v_1' that was produced by evaluation of e_1 under σ and ζ, and the value assignment $add(\sigma, x, v_1')$, such that v was produced by evaluation of e_2 under $add(\sigma, x, v_1')$ and ζ. From case 11ii we know $add(\sigma, x, v_1')$ is consistent with $\Gamma[x \mapsto \tau']$, so, by case 11iv, $v \in [\![\tau]\!]$.

12. Suppose the last rule used to derive $\Gamma, \Theta \vdash e : \tau$ was Rule T.15. Then $e = f(e_1, \ldots, e_n)$, with $\Theta(f) = \tau_1 \times \ldots \times \tau_n \to \tau_{out}$, and $\tau = \tau_{out}$. We can apply the induction hypothesis to $\Gamma, \Theta \vdash e_i : \tau_i$, for $i \in \{1, \ldots, n\}$, so

    (i) $\exists\, v_i \in \mathcal{V} \colon \sigma, \zeta \models e_i \Rightarrow v_i$;

    (ii) for all $v_i \in \mathcal{V}$, if $\sigma, \zeta \models e_i \Rightarrow v_i$ then $v_i \in [\![\tau_i]\!]$.

(a) For $i \in \{1, \ldots, n\}$, take v_i from case 12i. We know ζ is consistent with Θ, and, from case 12ii, $v_i \in [\![\tau_i]\!]$. By Definition 2.15, $\zeta(f)$ is a total relation, so there must be a $w \in [\![\tau_{out}]\!]$, such that $(v_1, \ldots, v_n, w) \in \zeta(f)$. We apply Rule V.17, therefore $v = w$.

(b) Let $v \in \mathcal{V}$, such that $\sigma, \zeta \models f(e_1, \ldots, e_n) \Rightarrow v$. The only rule that could have been last applied is Rule V.17, so, for $i \in \{1, \ldots, n\}$, there is a value v'_i, such that v'_i was produced by evaluation of e_i under σ and ζ, and $(v'_i, \ldots, v'_n, v) \in \zeta(f)$. Since ζ is consistent with Θ, $v \in [\![\tau_{out}]\!]$.

□

3

Towards a dataflow repository

3.1 Storing past executions of NRC dataflows

In a dataflow repository, we want to keep sufficient information about the different executions we have performed of each dataflow. To achieve this, it is not always enough to just keep the input values. Indeed, if non-deterministic service functions are used in a dataflow, merely rerunning the dataflow on the same inputs may not produce the same result as before. It is also not sufficient to keep only the final result of every execution in addition to the input values. When the final result value is vast and complex, it is desirable to be able to track how a particular subvalue of that result was produced during an execution. Again, as before, merely rerunning the dataflow will not do. It is obvious that we need to keep, for each execution of a dataflow, the output of each called service function in addition to the inputs of the dataflow.

Often some other intermediate result values, produced by evaluation of subexpressions other than service functions, are also important. Those intermediate results may help with debugging the dataflow during the design phase, or may enable performing partial reruns of the dataflow. We can naturally represent this information as triples consisting of an identification of executed subexpression, the used inputs, and the corresponding result. A set containing such triples, forms a log of an execution of a dataflow, or, in other words, a past "run" of a dataflow.

Eventually, the designer of a dataflow may indicate during design for which subexpressions the information should be stored in the repository, depending on the application of the dataflow. If it is desirable for the users to have access to complete executions of the dataflow, then at least the inputs of the dataflow, and the intermediate result values of all service functions that are known, or suspected, to be non-deterministic,[*] must be stored.

Therefore, in the next subsection, we define Rules R.1–R.17 that produce *a run of an NRC expression, on a chosen value assignment and function assignment.* The resulting set contains triples with information about *all* values produced during evaluation of that expression. As previously mentioned, *not all* these triples must be stored. Moreover, the same rules can be used to reconstruct a past execution from stored triples, as long as the stored triples provide adequate information.

Caveat 3.1. The complete, or as detailed as possible, record of a past execution of some workflow, applied to certain input, and leading to a final result, is commonly called the (workflow) *provenance* [MCF$^+$11] or *retrospective provenance* [FKSS08] of that result. In the context of our model, we refer to such a record as a *run*.

3.1.1 Run semantics of an NRC expression

Let e be an NRC expression, σ a value assignment over $FV(e)$, and ζ a function assignment over $SN(e)$.

We represent a *run R of e under σ and ζ* as a set consisting of triples. Each triple is of the form (Φ', σ', v'), where Φ' is an occurrence of subexpression e' of e, i.e., $\Phi' \leftarrow\!\circ\, e$,[†] and where $\sigma', \zeta \models e' \Rightarrow v'$ holds. In particular, if v is the final result value of evaluating e under σ and ζ in R, then R always contains the triple $([e], \sigma, v)$.

Recall that there can be more than one run for a given expression e, given a value assignment σ, and a function assignment ζ, due to the possibly non-deterministic nature of the service functions used in ζ. In general, there is at least one run for each possible result value of e under σ and ζ (Theorem 3.5).

The following Rules R.1–R.17 formally define *a run R of e under σ and ζ*. The rules infer judgements of the form $\sigma, \zeta \approx\!\!\models e \Rightarrow R$, meaning "$R$ is a possible run resulting from evaluation of e under σ and ζ". These rules also define *result*(R): the final result value of run R.

We intensively use the prefixing operation defined in Eq. A.1. The rules should be read with the operator "$\cdot$" having a higher precedence than the operator

[*]Even deterministic service functions may fail (due to, e.g., connection failure, or a non-compatible change to the invoked external service), and recording the failure may be advisable in some cases. From this point of view, all service functions are non-deterministic.

[†]Remember, we represent an occurrence of a subexpression by a subexpression path ending in that particular subexpression (Definition 2.20)

"$\cup$", for example, $[e,1]\cdot R_1 \cup [e,2]\cdot R_2$ should be read as $([e,1]\cdot R_1)\cup([e,2]\cdot R_2)$.

$$\frac{\mathtt{a} \in \mathcal{A} \qquad R := \{([\mathtt{a}],\, \sigma,\, \mathtt{a})\}}{\sigma, \zeta \approx\!\!\!\models \mathtt{a} \Rightarrow R \qquad \mathit{result}(R) \overset{\mathit{def}}{=} \mathtt{a}}\ \text{R.1}$$

$$\frac{x \in \mathcal{X} \qquad R := \{([x],\, \sigma,\, \mathit{get}(\sigma, x))\}}{\sigma, \zeta \approx\!\!\!\models x \Rightarrow R \qquad \mathit{result}(R) \overset{\mathit{def}}{=} \mathit{get}(\sigma, x)}\ \text{R.2}$$

$$\frac{R := \{([\varnothing],\, \sigma,\, \emptyset)\}}{\sigma, \zeta \approx\!\!\!\models \varnothing \Rightarrow R \qquad \mathit{result}(R) \overset{\mathit{def}}{=} \emptyset}\ \text{R.3}$$

$$\frac{\begin{array}{c} e = \{e'\} \\ \sigma, \zeta \approx\!\!\!\models e' \Rightarrow R' \qquad v := \{\mathit{result}(R')\} \qquad R := [e]\cdot R' \cup \{([e],\, \sigma,\, v)\} \end{array}}{\sigma, \zeta \approx\!\!\!\models e \Rightarrow R \qquad \mathit{result}(R) \overset{\mathit{def}}{=} v}\ \text{R.4}$$

$$\frac{\begin{array}{c} e = (e_1 \cup e_2) \\ \sigma, \zeta \approx\!\!\!\models e_1 \Rightarrow R_1 \qquad \sigma, \zeta \approx\!\!\!\models e_2 \Rightarrow R_2 \qquad v := \mathit{result}(R_1) \cup \mathit{result}(R_2) \\ R := [e,1]\cdot R_1 \cup [e,2]\cdot R_2 \cup \{([e],\, \sigma,\, v)\} \end{array}}{\sigma, \zeta \approx\!\!\!\models e \Rightarrow R \qquad \mathit{result}(R) \overset{\mathit{def}}{=} v}\ \text{R.5}$$

$$\frac{\begin{array}{c} e = \bigcup e' \\ \sigma, \zeta \approx\!\!\!\models e' \Rightarrow R' \qquad v := \bigcup \mathit{result}(R') \qquad R := [e]\cdot R' \cup \{([e],\, \sigma,\, v)\} \end{array}}{\sigma, \zeta \approx\!\!\!\models e \Rightarrow R \qquad \mathit{result}(R) \overset{\mathit{def}}{=} v}\ \text{R.6}$$

Let us explain Rule R.6. We see that, in order to be able to derive a possible run R of e on given σ and ζ, we must first derive a possible run R' for e' on the same σ and ζ. From this particular R', we construct a final result value v for e, and a run R of which v is the final result value. This R is one of the possible runs of e on σ and ζ, in particular the one that has R' as its *subrun* which, as we show in Section 4.2.1, can be found in R by subexpression path $[e; e']$ and σ.

$$\frac{\begin{array}{c} e = \langle l_1 : e_1, \ldots, l_n : e_n \rangle \qquad \forall i \in \{1, \ldots, n\} \colon \sigma, \zeta \approx\!\!\!\models e_i \Rightarrow R_i \\ v := \langle l_1 : \mathit{result}(R_1), \ldots, l_n : \mathit{result}(R_n) \rangle \\ R := \left(\bigcup_{i \in \{1,\ldots,n\}} [e, l_i] \cdot R_i \right) \cup \{([e],\, \sigma,\, v)\} \end{array}}{\sigma, \zeta \approx\!\!\!\models e \Rightarrow R \qquad \mathit{result}(R) \overset{def}{=} v} \text{ R.7}$$

$$\frac{\begin{array}{c} e = e'.l \qquad \sigma, \zeta \approx\!\!\!\models e' \Rightarrow R' \\ \mathit{result}(R') = \langle \ldots, l \colon v, \ldots \rangle \qquad R := [e] \cdot R' \cup \{([e],\, \sigma,\, v)\} \end{array}}{\sigma, \zeta \approx\!\!\!\models e \Rightarrow R \qquad \mathit{result}(R) \overset{def}{=} v} \text{ R.8}$$

$$\frac{\begin{array}{c} e = \textsf{for } x \textsf{ in } e_1 \textsf{ return } e_2 \\ \sigma, \zeta \approx\!\!\!\models e_1 \Rightarrow R' \qquad \mathit{result}(R') = \{w_1, \ldots, w_n\} \text{ with all distinct } w_i \\ \forall i \in \{1, \ldots, n\} \colon \mathit{add}(\sigma, x, w_i), \zeta \approx\!\!\!\models e_2 \Rightarrow R_i \\ v := \{\mathit{result}(R_1), \ldots, \mathit{result}(R_n)\} \\ R := [e, 1] \cdot R' \cup \left(\bigcup_{i \in \{1,\ldots,n\}} [e, 2] \cdot R_i \right) \cup \{([e],\, \sigma,\, v)\} \end{array}}{\sigma, \zeta \approx\!\!\!\models e \Rightarrow R \qquad \mathit{result}(R) \overset{def}{=} v} \text{ R.9}$$

Let us explain Rule R.9. We see that we must first derive a possible run R' for e_1 on the same σ and ζ. We then use the elements of the result value of R': for each element w_i, we extend σ by adding the pair (x, w_i), and use the new value assignment together with ζ to derive a possible run R_i for e_2. From all these runs of e_2, we construct a final result value v for e, and then we use R' and all the runs of e_2 to construct a run R of e, of which v is the final result value. This run R is, again, one of the possible runs of e under σ and ζ, in particular the one that has R' and all the runs of e_2 as its subruns. We can find R' in R by $[e; 1; e_1]$ and σ, and we can find each R_i by $[e; 2; e_2]$ and $\mathit{add}(\sigma, x, w_i)$.

$$\frac{\begin{array}{c} e = (e_1 = e_2) \\ \sigma, \zeta \approx\!\!\!\models e_1 \Rightarrow R_1 \qquad \sigma, \zeta \approx\!\!\!\models e_2 \Rightarrow R_2 \qquad \mathit{result}(R_1) = \mathit{result}(R_2) \\ v := \texttt{true} \qquad R := [e, 1] \cdot R_1 \cup [e, 2] \cdot R_2 \cup \{([e],\, \sigma,\, v)\} \end{array}}{\sigma, \zeta \approx\!\!\!\models e \Rightarrow R \qquad \mathit{result}(R) \overset{def}{=} v} \text{ R.10}$$

$$\frac{\begin{array}{c} e = (e_1 = e_2) \\ \sigma, \zeta \approx\!\!\models e_1 \Rightarrow R_1 \qquad \sigma, \zeta \approx\!\!\models e_2 \Rightarrow R_2 \qquad \mathit{result}(R_1) \neq \mathit{result}(R_2) \\ v := \texttt{false} \qquad R := [e,1] \cdot R_1 \cup [e,2] \cdot R_2 \cup \{([e],\, \sigma,\, v)\} \end{array}}{\sigma, \zeta \approx\!\!\models e \Rightarrow R \qquad \mathit{result}(R) \overset{def}{=} v} \;\text{R.11}$$

$$\frac{\begin{array}{c} e = (e' = \varnothing) \qquad \sigma, \zeta \approx\!\!\models e' \Rightarrow R' \\ \mathit{result}(R') = \emptyset \qquad v := \texttt{true} \qquad R := [e] \cdot R' \cup \{([e],\, \sigma,\, v)\} \end{array}}{\sigma, \zeta \approx\!\!\models e \Rightarrow R \qquad \mathit{result}(R) \overset{def}{=} v} \;\text{R.12}$$

$$\frac{\begin{array}{c} e = (e' = \varnothing) \qquad \sigma, \zeta \approx\!\!\models e' \Rightarrow R' \\ \mathit{result}(R') \neq \emptyset \qquad v := \texttt{false} \qquad R := [e] \cdot R' \cup \{([e],\, \sigma,\, v)\} \end{array}}{\sigma, \zeta \approx\!\!\models e \Rightarrow R \qquad \mathit{result}(R) \overset{def}{=} v} \;\text{R.13}$$

$$\frac{\begin{array}{c} e = \textsf{if } e_0 \textsf{ then } e_1 \textsf{ else } e_2 \\ \sigma, \zeta \approx\!\!\models e_0 \Rightarrow R_0 \qquad \mathit{result}(R_0) = \texttt{true} \qquad \sigma, \zeta \approx\!\!\models e_1 \Rightarrow R_1 \\ v := \mathit{result}(R_1) \qquad R := [e,0] \cdot R_0 \cup [e,1] \cdot R_1 \cup \{([e],\, \sigma,\, v)\} \end{array}}{\sigma, \zeta \approx\!\!\models e \Rightarrow R \qquad \mathit{result}(R) \overset{def}{=} v} \;\text{R.14}$$

$$\frac{\begin{array}{c} e = \textsf{if } e_0 \textsf{ then } e_1 \textsf{ else } e_2 \\ \sigma, \zeta \approx\!\!\models e_0 \Rightarrow R_0 \qquad \mathit{result}(R_0) = \texttt{false} \qquad \sigma, \zeta \approx\!\!\models e_2 \Rightarrow R_2 \\ v := \mathit{result}(R_2) \qquad R := [e,0] \cdot R_0 \cup [e,2] \cdot R_2 \cup \{([e],\, \sigma,\, v)\} \end{array}}{\sigma, \zeta \approx\!\!\models e \Rightarrow R \qquad \mathit{result}(R) \overset{def}{=} v} \;\text{R.15}$$

$$\frac{\begin{array}{c} e = \textsf{let } x := e_1 \textsf{ in } e_2 \\ \sigma, \zeta \approx\!\!\models e_1 \Rightarrow R_1 \qquad \mathit{add}(\sigma, x, \mathit{result}(R_1)), \zeta \approx\!\!\models e_2 \Rightarrow R_2 \\ v := \mathit{result}(R_2) \qquad R := [e,1] \cdot R_1 \cup [e,2] \cdot R_2 \cup \{([e],\, \sigma,\, v)\} \end{array}}{\sigma, \zeta \approx\!\!\models e \Rightarrow R \qquad \mathit{result}(R) \overset{def}{=} v} \;\text{R.16}$$

$$\frac{\begin{array}{c} e = f(e_1, \dots, e_n) \qquad \forall i \in \{1, \dots, n\} \colon \sigma, \zeta \mathrel{|}\!\approx e_i \Rightarrow R_i \\ (\mathit{result}(R_1), \dots, \mathit{result}(R_n), v) \in \zeta(f) \\ R := \left(\bigcup_{i \in \{1,\dots,n\}} [e, i] \cdot R_i\right) \cup \{([e],\ \sigma,\ v)\} \end{array}}{\sigma, \zeta \mathrel{|}\!\approx e \Rightarrow R \qquad \mathit{result}(R) \overset{\mathit{def}}{=} v} \text{ R.17}$$

Example 3.2. We execute the following NRC expression: $x \cup \{y.r\}$. Let $e = x \cup e_1$, $e_1 = \{y.r\}$, and $e_2 = y.r$. Let $\sigma = [(x, in_1), (y, in_2)]$, with

$$\begin{aligned} in_1 &= \{\langle a\colon 1, b\colon 1\rangle, \langle a\colon 3, b\colon 9\rangle, \langle a\colon 5, b\colon 25\rangle\}, \\ in_2 &= \langle k\colon \mathsf{odd}, r\colon \langle a\colon 5, b\colon 25\rangle\rangle. \end{aligned}$$

The only run R of e under given σ and empty ζ is[‡]

$$\begin{aligned} R = \{\, &([e, 1, x],\ \sigma,\ in_1), \\ &([e, 2, e_1, e_2, y],\ \sigma,\ in_2), \\ &([e, 2, e_1, e_2],\ \sigma,\ v_p), \\ &([e, 2, e_1],\ \sigma,\ v_s), \\ &([e],\ \sigma,\ out)\,\} \end{aligned}$$

with $v_p = \langle a\colon 5, b\colon 25\rangle$, $v_s = \{\langle a\colon 5, b\colon 25\rangle\}$, and

$$out = \{\langle a\colon 1, b\colon 1\rangle, \langle a\colon 3, b\colon 9\rangle, \langle a\colon 5, b\colon 25\rangle\}.$$

□

Example 3.3. We execute the following NRC expression:

$$\mathsf{for}\ x\ \mathsf{in}\ y\ \mathsf{return}\ \langle b\colon x.b, c\colon f(x.a)\rangle.$$

Let $e = \mathsf{for}\ x\ \mathsf{in}\ y\ \mathsf{return}\ e_1$, $e_1 = \langle b\colon e_2, c\colon e_3\rangle$, $e_2 = x.b$, $e_3 = f(e_4)$, and $e_4 = x.a$. Let $\sigma = [(y, in)]$, with $in = \{in_1, in_2, in_3\}$, and

$$\begin{aligned} in_1 &= \langle a\colon 2, b\colon 4\rangle, \\ in_2 &= \langle a\colon 5, b\colon 2\rangle, \\ in_3 &= \langle a\colon 5, b\colon 4\rangle. \end{aligned}$$

[‡]There are no service names in e, so $\zeta = \emptyset$.

Let $\{(2,1),(5,1),(5,7)\} \subseteq \zeta(f)$. A possible run R of e under given σ and ζ is:

$$\begin{aligned}
R = \{\, & ([e,1,y],\, \sigma,\, in),\\
& ([e,2,e_1,b,e_2,x],\, add(\sigma,x,in_1),\, in_1),\\
& ([e,2,e_1,b,e_2,x],\, add(\sigma,x,in_2),\, in_2),\\
& ([e,2,e_1,b,e_2,x],\, add(\sigma,x,in_3),\, in_3),\\
& ([e,2,e_1,b,e_2],\, add(\sigma,x,in_1),\, b_1),\\
& ([e,2,e_1,b,e_2],\, add(\sigma,x,in_2),\, b_2),\\
& ([e,2,e_1,b,e_2],\, add(\sigma,x,in_3),\, b_3),\\
& ([e,2,e_1,c,e_3,1,e_4,x],\, add(\sigma,x,in_1),\, in'_1),\\
& ([e,2,e_1,c,e_3,1,e_4,x],\, add(\sigma,x,in_2),\, in'_2),\\
& ([e,2,e_1,c,e_3,1,e_4,x],\, add(\sigma,x,in_3),\, in'_3),\\
& ([e,2,e_1,c,e_3,1,e_4],\, add(\sigma,x,in_1),\, a_1),\\
& ([e,2,e_1,c,e_3,1,e_4],\, add(\sigma,x,in_2),\, a_2),\\
& ([e,2,e_1,c,e_3,1,e_4],\, add(\sigma,x,in_3),\, a_3),\\
& ([e,2,e_1,c,e_3],\, add(\sigma,x,in_1),\, w_1),\\
& ([e,2,e_1,c,e_3],\, add(\sigma,x,in_2),\, w_2),\\
& ([e,2,e_1,c,e_3],\, add(\sigma,x,in_3),\, w_3),\\
& ([e,2,e_1],\, add(\sigma,x,in_1),\, v_1),\\
& ([e,2,e_1],\, add(\sigma,x,in_2),\, v_2),\\
& ([e,2,e_1],\, add(\sigma,x,in_3),\, v_3),\\
& ([e],\, \sigma,\, out)\,\}
\end{aligned}$$

with $b_1 = b_3 = 4$, $b_2 = 2$, $in'_1 = in_1$, $in'_2 = in_2$, $in'_3 = in_3$, $a_1 = 2$, $a_2 = a_3 = 5$, $w_1 = w_3 = 1$, $w_2 = 7$, $v_1 = \langle b\colon 4, c\colon 1\rangle$, $v_2 = \langle b\colon 2, c\colon 7\rangle$, $v_3 = \langle b\colon 4, c\colon 1\rangle$, and $out = \{\langle b\colon 4, c\colon 1\rangle, \langle b\colon 2, c\colon 7\rangle\}$. □

Note that in the third component of each run-triple, instead of a complex value, we use a unique identifier for that value (unique in the scope of the run). For a complex value, the unique identifier helps to locate the correct triple the value belongs to. In the following, we continue to use these examples to illustrate other concepts, and the identifiers are useful as a visual aide. (In Proposition 3.10 we prove that we can uniquely identify a complex value in a run, without explicit identifiers).

The following proposition can be proven by straightforward comparison of each rule from Rules R.1–R.17 with the corresponding rule from Rules V.1–V.17.

Proposition 3.4. *If* $\sigma, \zeta \approx\!\!\!\models e \Rightarrow R$, *then* $\sigma, \zeta \models e \Rightarrow \mathit{result}(R)$.

Theorem 3.5 (Soundness). *Let Γ be a type assignment over $FV(e)$, and Θ a signature assignment over $SN(e)$. If σ is consistent with Γ, and ζ is consistent with Θ, and $\Gamma, \Theta \vdash e : \tau$, then the following holds:*

(a) there is a run R, such that $\sigma, \zeta \approx e \Rightarrow R$;

(b) for each run R, if $\sigma, \zeta \approx e \Rightarrow R$ then $result(R) \in [\![\tau]\!]$.

Proof. Part b of the theorem follows directly from Proposition 3.4 and Theorem 2.30.

We prove part a by induction on the structure of e. The base case involves a constant, a variable, or an empty-set expression.

If $e = \mathtt{a}$, we apply Rule R.1, thus $R = \{([\mathtt{a}], \sigma, \mathtt{a})\}$. If $e = x$, we apply Rule R.2, thus $R = \{([x], \sigma, get(\sigma, x))\}$. If $e = \varnothing$, we apply Rule R.3, thus $R = \{([\varnothing], \sigma, \emptyset)\}$.

Let us assume part a of the theorem holds for expressions of height n.[§] We now prove it also holds for expressions of height $n + 1$.

1. If e is of the form

   (i) $\{e'\}$,

   (ii) $\bigcup e'$, or

   (iii) $e'.l$,

   then we can only consider (i) Rule R.4; (ii) Rule R.6; or (iii) Rule R.8. We can apply the induction hypothesis to e', so there is a run R', such that $\sigma, \zeta \approx e' \Rightarrow R'$, with $result(R')$ of the same type as e'. Let (i) $v = \{result(R')\}$; (ii) $v = \bigcup result(R')$; (iii) $result(R') = \langle \ldots, l \colon v, \ldots \rangle$. Let $R = [e] \cdot R' \cup \{([e], \sigma, v)\}$. By the corresponding rule, $\sigma, \zeta \approx e \Rightarrow R$.

2. If e is of the form

   (i) $e_1 \cup e_2$,

   (ii) $\langle e_1 : l_1, \ldots, e_n : l_n \rangle$, or

   (iii) $f(e_1, \ldots, e_n)$,

   then we can only consider (i) Rule R.5; (ii) Rule R.7; or (iii) Rule R.17. Let (i) $I = \{1, 2\}$; (ii–iii) $I = \{1, \ldots, n\}$. For $i \in I$, we can apply the induction hypothesis to e_i, so there is a run R_i such that $\sigma, \zeta \approx e_i \Rightarrow R_i$, with $result(R_i)$ of the same type as e_i. Let

[§]By "height" of an expression e we mean the height of the syntax tree of e.

(i) $v = \mathit{result}(R_1) \cup \mathit{result}(R_2)$, and, for $i \in I$, $m_i = i$;

(ii) $v = \langle l_1 \colon \mathit{result}(R_1), \dots, l_n \colon \mathit{result}(R_n)\rangle$, and, for $i \in I$, $m_i = l_i$;

(iii) $(\mathit{result}(R_1), \dots, \mathit{result}(R_n), v) \in \zeta(f)$, and, for $i \in I$, $m_i = i$.

Let $R = \bigcup_{i\in I} [e, m_i] \cdot R_i \cup \{([e], \sigma, v)\}$. By the corresponding rule, $\sigma, \zeta \approx\!\!\models e \Rightarrow R$.

3. If $e =$ for x in e_1 return e_2, then we can only consider Rule R.9. Since e is well-typed, there is a type τ_1 such that $e_1 \colon \{\tau_1\}$. We can apply the induction hypothesis to e_1, so there is a run R_1 such that $\sigma, \zeta \approx\!\!\models e_1 \Rightarrow R_1$, with $\mathit{result}(R_1) \in [\![\{\tau_1\}]\!]$. Since, for each $w \in \mathit{result}(R_1)$, $\Gamma[x \mapsto \tau_1]$ is consistent with $\mathit{add}(\sigma, x, w)$, we can apply the induction hypothesis to e_2, so there is a run R_w such that $\mathit{add}(\sigma, x, w), \zeta \approx\!\!\models e_2 \Rightarrow R_w$. Let $v = \{\mathit{result}(R_w) \mid w \in \mathit{result}(R_1)\}$ and

$$R = [e,1] \cdot R_1 \cup \bigcup_{w\in \mathit{result}(R_1)} [e,2] \cdot R_w \cup \{([e], \sigma, v)\}.$$

By Rule R.9, $\sigma, \zeta \approx\!\!\models e \Rightarrow R$.

4. If $e = (e_1 = e_2)$, then we can only consider Rules R.10–R.11. We can apply the induction hypothesis to both e_1 and e_2, so there are runs R_1 and R_2, such that, for $i \in \{1, 2\}$, $\sigma, \zeta \approx\!\!\models e_i \Rightarrow R_i$, with $\mathit{result}(R_i)$ of the same type as e_i. Let $R = [e, 1] \cdot R_1 \cup [e, 2] \cdot R_2 \cup \{([e], \sigma, v)\}$, with $v = \texttt{true}$ if $\mathit{result}(R_1) = \mathit{result}(R_2)$, and $v = \texttt{false}$ if $\mathit{result}(R_1) \neq \mathit{result}(R_2)$. By Rule R.10 and Rule R.11, respectively, $\sigma, \zeta \approx\!\!\models e \Rightarrow R$.

5. If $e = (e' = \varnothing)$, then we can only consider Rules R.12–R.13. We can apply the induction hypothesis to e', so there is a run R', such that $\sigma, \zeta \approx\!\!\models e' \Rightarrow R'$, with $\mathit{result}(R')$ of the same type as e'. Let $R = [e] \cdot R' \cup \{([e], \sigma, v)\}$, with $v = \texttt{true}$ if $\mathit{result}(R') = \emptyset$, and $v = \texttt{false}$ if $\mathit{result}(R') \neq \emptyset$. By Rule R.12 and Rule R.13, respectively, $\sigma, \zeta \approx\!\!\models e \Rightarrow R$.

6. If $e =$ if e_0 then e_1 else e_2, then we can only consider Rules R.14–R.15. For $i \in \{0, 1, 2\}$, we can apply the induction hypothesis to e_i, so there is a run R_i such that $\sigma, \zeta \approx\!\!\models e_i \Rightarrow R_i$, with $\mathit{result}(R_i)$ of the same type as e_i. Let $R = [e, 0] \cdot R_0 \cup [e, j] \cdot R_j \cup \{([e], \sigma, \mathit{result}(R_j))\}$, with $j = 1$ if $\mathit{result}(R_0) = \texttt{true}$, and $j = 2$ if $\mathit{result}(R_0) = \texttt{false}$. By Rule R.14 and Rule R.15, respectively, $\sigma, \zeta \approx\!\!\models e \Rightarrow R$.

7. If $e =$ let $x := e_1$ in e_2, then we can only consider Rule R.16. Since e is well-typed, there is a type τ' such that $e_1 \colon \tau'$. We can apply the induction hypothesis to e_1, so there is a run R_1 such that $\sigma, \zeta \approx\!\!\models e_1 \Rightarrow R_1$, with $\mathit{result}(R_1) \in [\![\tau']\!]$. Since $\Gamma[x \mapsto \tau']$ is consistent with

$add(\sigma, x, result(R_1))$, we can apply the induction hypothesis to e_2, so there is a run R_2 such that $add(\sigma, x, result(R_1)), \zeta \approx e_2 \Rightarrow R_2$. Let $R = [e,1] \cdot R_1 \cup [e,2] \cdot R_2 \cup \{([e], \sigma, result(R_2))\}$. By Rule R.16, $\sigma, \zeta \approx e \Rightarrow R$.

□

Next, after three auxiliary lemma's, we show that each triple in a run can be uniquely identified.

Definition 3.6 (Subexpression invocation). Let $\sigma, \zeta \approx e \Rightarrow R$ and $(\Phi', \sigma', v') \in R$. We call (Φ', σ') a *subexpression invocation of* R. We use $SI(R)$ to denote the set of all subexpression invocations of R.

Lemma 3.7. *Let $\sigma, \zeta \approx e \Rightarrow R$. Then there is only one triple in R having $[e]$ as its first component, i.e., the triple $([e], \sigma, result(R))$. Moreover, there are no other triples with a trivial subexpression path in the first component.*

Proof. A simple inspection of the rules in Section 3.1.1 is sufficient. Indeed, in each rule, the only triple with a trivial subexpression path in the first component that is added to run R is exactly $([e], \sigma, result(R))$. □

Lemma 3.8. *Let $\sigma, \zeta \approx e \Rightarrow R$ and $(\Phi', \sigma', v') \in R$. Then*

1. *Φ' is a subexpression path of e;*
2. *σ is a prefix of σ'.*

Proof. A simple inspection of the rules in Section 3.1.1 is sufficient.

1. Indeed, the sequence in the first component of each triple is constructed according to Definition 2.20.
2. The only rules where an occurrence of a subexpression of e is evaluated under another value assignment than σ, are Rule R.9 for an for-expression, and Rule R.16 for an let-expression. Value assignments used in these rules, by construction, clearly have σ as a prefix.

□

Lemma 3.9. *Let $\sigma, \zeta \approx e_1 \Rightarrow R'$. Let $result(R') = \{w_1, \ldots, w_n\}$, with all distinct w_i. For $i \in \{1, \ldots, n\}$, let $add(\sigma, x, w_i), \zeta \approx e_2 \Rightarrow R_i$. Then, for $i, j \in \{1, \ldots, n\}$, R_i and R_j are disjoint if $i \neq j$.*

Proof. For each $i \in \{1, \ldots, n\}$ and $(\Phi', \sigma', v') \in R_i$, by Lemma 3.8, $add(\sigma, x, w_i)$ is a prefix of σ'. Since all values w_i are distinct, also all $add(\sigma, x, w_i)$ are distinct. Hence, all R_i are mutually disjoint. □

Proposition 3.10. *Let $\sigma, \zeta \approx e \Rightarrow R$ and $(\Phi, \sigma') \in SI(R)$. Then (Φ, σ') corresponds to exactly one triple in R.*

Proof. We prove the lemma by induction on the structure of e.

If e is a constant (Rule R.1), a variable (Rule R.2), or an empty-set expression (Rule R.3), then R is the singleton $\{([e], \sigma, result(R))\}$ and $(\Phi, \sigma') = ([e], \sigma)$. Clearly, $([e], \sigma)$ corresponds to exactly one triple in R.

Let us assume the lemma holds for expressions of height n. We now prove it also holds for expressions of height $n + 1$.

1. If e is of the form
   (a) $\{e'\}$ (Rule R.4),
   (b) $\bigcup e'$ (Rule R.6),
   (c) $e'.l$ (Rule R.8), or
   (d) $e' = \varnothing$ (Rules R.12–R.13),

   then it follows from the corresponding rules, that there is a run R' for e', such that
   $$R = [e] \cdot R' \cup \{([e], \sigma, result(R))\}.$$
   Since $(\Phi, \sigma') \in SI(R)$, by Definition 3.6, there is a value v' such that $(\Phi, \sigma', v') \in R$.

   Let $(\Phi, \sigma', v') \in \{([e], \sigma, result(R))\}$. Then $(\Phi, \sigma', v') = ([e], \sigma, result(R))$. Hence, by Lemma 3.7, $([e], \sigma)$ corresponds to exactly one triple in R.

   Let $(\Phi, \sigma', v') \in [e] \cdot R'$. Then, by Lemma 3.7, $\Phi = [e] \cdot \Psi$, with a non-empty Ψ. Therefore $([e] \cdot \Psi, \sigma', v') \notin \{([e], \sigma, result(R))\}$. Moreover, $([e] \cdot \Psi, \sigma', v') \in [e] \cdot R'$ implies $(\Psi, \sigma', v') \in R'$, so $(\Psi, \sigma') \in SI(R')$. We can now apply the induction hypothesis to R', thus (Ψ, σ') corresponds to exactly one triple in R'. Hence $([e] \cdot \Psi, \sigma')$ corresponds to exactly one triple in R.

2. Expression e is of the form
   (a) $e_1 \cup e_2$ (Rule R.5),
   (b) $e_1 = e_2$ (Rules R.10–R.11),

(c) let $x := e_1$ in e_2 (Rule R.16),

(d) $e =$ if e_0 then e_1 else e_2 (Rules R.14–R.15),

(e) $\langle l_1 : e_1, \ldots, l_n : e_n \rangle$ (Rule R.7), or

(f) $f(e_1, \ldots, e_n)$ (Rule R.17).

Let

(a–c) $I = \{1, 2\}$ and, for $i \in I$, $m_i = i$;

(d) $I = \{0, 1\}$ if $\mathit{result}(R_0) = \mathtt{true}$; $I = \{0, 2\}$ if $\mathit{result}(R_0) = \mathtt{false}$; and, for $i \in I$, $m_i = i$;

(e) $I = \{1, \ldots, n\}$ and, for $i \in I$, $m_i = l_i$;

(f) $I = \{1, \ldots, n\}$ and, for $i \in I$, $m_i = i$.

Then, by the corresponding rules, there are runs R_i for e_i, for $i \in I$, such that

$$R = \bigcup_{i \in I} [e, m_i] \cdot R_i \,\cup\, \{([e], \sigma, \mathit{result}(R))\}\,.$$

Since $(\Phi, \sigma') \in \mathit{SI}(R)$, by Definition 3.6, there is a value v' such that $(\Phi, \sigma', v') \in R$.

Let $(\Phi, \sigma', v') \in \{([e], \sigma, \mathit{result}(R))\}$. Then $(\Phi, \sigma', v') = ([e], \sigma, \mathit{result}(R))$. Hence, by Lemma 3.7, $([e], \sigma)$ corresponds to exactly one triple in R.

Let $j \in I$ and $(\Phi, \sigma', v') \in [e, m_j] \cdot R_j$. Then Φ is of the form $[e, m_j] \cdot \Psi$, for some Ψ. Therefore $([e, m_j] \cdot \Psi, \sigma', v') \notin \{([e], \sigma, \mathit{result}(R))\}$. Also $([e, m_j] \cdot \Psi, \sigma', v') \notin [e, m_k] \cdot R_k$, for $k \in I \setminus \{j\}$, because the prefix $[e, m_j]$ does not match. Moreover, $([e, m_j] \cdot \Psi, \sigma', v') \in [e, m_j] \cdot R_j$ implies $(\Psi, \sigma', v') \in R_j$, so $(\Psi, \sigma') \in \mathit{SI}(R_j)$. We can now apply the induction hypothesis to R_j, thus (Ψ, σ') corresponds to exactly one triple in R_j. Hence $([e, m_j] \cdot \Psi, \sigma')$ corresponds to exactly one triple in R.

3. If $e =$ for x in e_1 return e_2, then it follows from Rule R.9, that there is a run R_1 for e_1, and for $w \in \mathit{result}(R_1)$, there are runs R_w for e_2, such that

$$R = [e, 1] \cdot R_1 \,\cup\, (\textstyle\bigcup_w [e, 2] \cdot R_w) \,\cup\, \{([e], \sigma, \mathit{result}(R))\}\,.$$

Since $(\Phi, \sigma') \in \mathit{SI}(R)$, by Definition 3.6, there is a value v' such that $(\Phi, \sigma', v') \in R$.

Let $(\Phi, \sigma', v') \in \{([e], \sigma, \mathit{result}(R))\}$. Then $(\Phi, \sigma', v') = ([e], \sigma, \mathit{result}(R))$. Hence, by Lemma 3.7, $([e], \sigma)$ corresponds to exactly one triple in R.

Let $(\Phi, \sigma', v') \in [e,1] \cdot R_1$. Then Φ is of the form $[e,1] \cdot \Psi$, for some Ψ. Therefore $([e,1] \cdot \Psi, \sigma', v') \notin \{([e], \sigma, \mathit{result}(R))\}$. Also $([e,1] \cdot \Psi, \sigma', v') \notin \bigcup_w [e,2] \cdot R_w$, because the prefix $[e,1]$ does not match. Moreover, $([e,1] \cdot \Psi, \sigma', v') \in [e,1] \cdot R_1$ implies $(\Psi, \sigma', v') \in R_1$, so $(\Psi, \sigma') \in \mathit{SI}(R_1)$. We can now apply the induction hypothesis to R_1, thus (Ψ, σ') corresponds to exactly one triple in R_1. Hence $([e,1] \cdot \Psi, \sigma')$ corresponds to exactly one triple in R.

Let $(\Phi, \sigma', v') \in \bigcup_w [e,2] \cdot R_w$. Then Φ is of the form $[e,2] \cdot \Psi$, for some Ψ. Therefore $([e,2] \cdot \Psi, \sigma', v') \notin \{([e], \sigma, \mathit{result}(R))\}$. Also $([e,2] \cdot \Psi, \sigma', v') \notin [e,1] \cdot R_1$, because the prefix $[e,2]$ does not match. From Lemma 3.9, we know that there is only one $u \in \mathit{result}(R_1)$, such that $(\Psi, \sigma', v') \in R_u$ and $(\Psi, \sigma') \in \mathit{SI}(R_u)$. We can now apply the induction hypothesis to R_u, thus (Ψ, σ') corresponds to exactly one triple in R_u. Hence $([e,2] \cdot \Psi, \sigma')$ corresponds to exactly one triple in R.

□

3.2 External services and subdataflows

Recall that, in a dataflow repository, we would like to store different dataflows together with their past executions. Each dataflow could be potentially executed for distinct value assignments (inputs), and even more, with different function assignments (services). Remember that a function assignment binds the service names occurring in the dataflow to service functions. Up till now we have simply stated that a service function from $[\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$ to $[\![\tau_{out}]\!]$ is a total relation on $[\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$. However, each service function stands for an actual, executable service. Therefore, we must provide a mechanism to bind the abstract services to executable ones, essentially providing a translation of the abstract specification into an executable dataflow. Here, we make a distinction between *external* and *internal* services.

A service is *external* when it is defined outside our system, e.g., an application that queries a public database, an application that executes a particular algorithm, or a library function providing data transformations that cannot be expressed in NRC. Such a service is often a "black box", when only incomplete information is available about the computation it provides. In short, an *external service* is *not* a dataflow. To access an external service, we use a service function as an interface.¶

¶We still require all service functions to be total. If a service function *IO* is used as an interface to an external service, then *IO* is most likely a wrapper relation for that external service. We briefly discuss constructing a wrapper relation in Section A.2.

Note that external services account for the non-determinism in our system. Indeed, a data entry of a public database may be updated, an algorithm may be revised, and a library function may contain some bugs that need addressing. In general, we cannot expect all external services to be deterministic, even if their execution is always successful.

Definition 3.11 (External services). Assume a countably infinite set $\mathcal{E}$ of *external service identifiers.* Assume a given mapping $sig\colon \mathcal{E} \to \mathcal{S}$, which assigns a signature to each external service ID. Assume a given mapping $func\colon \mathcal{E} \to \mathcal{F}$, which assigns a service function to each external service ID.

We require *sig* and *func* to be *consistent*, i.e.,

$$\forall\, s \in \mathcal{E}\colon sig(s) = \tau_1 \times \ldots \times \tau_n \to \tau_{out} \implies func(s) \subseteq [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!] \times [\![\tau_{out}]\!]\,.$$

For $s \in \mathcal{E}$, we call $(s, sig(s), func(s))$ the *external service identified by s.* When context allows, we simply use s to refer to $(s, sig(s), func(s))$.

Obviously, an *internal service* is a dataflow. The use of already existing dataflows has several benefits, e.g., it enables modular design of dataflows and allows for reuse of dataflows. In essence, all previously designed dataflows comprise a dataflow library.

Definition 3.12 (Dataflows). Assume a countably infinite set $\mathcal{D}$ of *dataflow identifiers*, with $\mathcal{D}$ and $\mathcal{E}$ disjoint.

Assume a given mapping $expr\colon \mathcal{D} \to \mathcal{NRC}$, which assigns an NRC expression to each dataflow ID. Assume a given mapping $types\colon \mathcal{D} \to \mathcal{TA}$, which assigns to each dataflow ID d a type assignment over $FV(expr(d))$. Assume a given mapping $signatures\colon \mathcal{D} \to \mathcal{S}$, which assigns to each dataflow ID d a signature assignment over $SN(expr(d))$.

For $d \in \mathcal{D}$, we call $(d, expr(d), types(d), signatures(d))$ the *dataflow identified by d.* We require $expr(d)$ to be well-typed under $types(d)$ and $signatures(d)$. When context allows, we use d to refer to its dataflow.

If we want to translate a dataflow into an executable one, we need to associate service names with external services or available dataflows, and subsequently construct a function assignment for the dataflow. When a service name is associated with a dataflow d, we refer to d as a *subdataflow.* Indeed, its execution is initiated from another dataflow, its *parent dataflow.* A complication that arises from the use of subdataflows, is that a subdataflow d may in turn contain service names, associated with other subdataflows, for which d becomes a parent dataflow. In order to avoid non-terminating executions, we must pay attention not to create cycles. All these issues are taken care of by a *binding tree*, and a function assignment derived from a binding tree.

3.2.1 Binding trees

Intuitively, a binding tree specifies, for the dataflow identified by its root label, say d_r, which service names in the NRC expression of d_r are bound to external services, and which to subdataflows. For these subdataflows, the binding tree again specifies a binding for their own service names, and so on. During this process, care must be taken to ensure that the NRC expression of the parent dataflow, and the expressions of all subdataflows remain well-typed.

Therefore, we first specify conditions under which a service name f with signature ϑ can be associated with an external service or a dataflow. An external service is suitable if the complex types in its signature, when used as a replacement for types in ϑ, allow for the application of the subtyping rule T.4.

Definition 3.13 (Signature replacement mapping). Let $\vartheta_1 = \tau_1 \times \ldots \times \tau_n \rightarrow \tau_{out}$ and $\vartheta_2 = \mu_1 \times \ldots \times \mu_m \rightarrow \mu_{out}$ be two signatures. We say that ϑ_2 is a *signature replacement candidate* for ϑ_1 if and only if the following holds:

1. $\forall\, i \in \{1, \ldots, m\}\, \exists\, j \in \{1, \ldots, n\} \colon \tau_j \preceq \mu_i$, and
2. $\tau_{out} \preceq \mu_{out}$.

If ϑ_2 is a signature replacement candidate for ϑ_1, then a *signature replacement mapping* for ϑ_1 by ϑ_2 is a function $rep\colon \{1, \ldots, m\} \rightarrow \{1, \ldots, n\}$, such that $\forall\, i \in \{1, \ldots, m\} \colon \tau_{rep(i)} \preceq \mu_i$.

Note 3.2.1. In the above definition m may not equal n. For example, if a service $g\colon \mathbf{Int} \rightarrow \mathbf{Int}$ is supposed to return the square of its input, we may bind it to external service `multiply`: $\mathbf{Int} \times \mathbf{Int} \rightarrow \mathbf{Int}$, with the following signature replacement mapping: $\{(1,1),(2,1)\}$. During execution, $g(x)$ will be replaced by `multiply`(x, x). Likewise, if a service g: File $\times$ **Database** $\rightarrow$ Report is supposed to perform a search in a database, and it is always called in the dataflow expression with a constant `db` specifying the database, then we may bind it to external service `searchDB`: File $\rightarrow$ Report, performing the required search in database `db`. The signature replacement mapping is $\{(1,1)\}$, and $g(x, \texttt{db})$ will be replaced by `searchDb`(x). Another example for $m < n$ is when a parameter is simply ignored. If a service g: **Database** $\times$ **Keyword** $\times$ **Quality** $\rightarrow$ $\{$**Image**$\}$ is supposed to retrieves images from the specified database, based on the given keyword and quality specification, then we may bind it to external service `keywordSearch`: **Database** $\times$ **Keyword** $\rightarrow$ $\{$**Image**$\}$, ignoring the quality specification (it may be the only external service available, or it searches in databases that only provide one quality). The signature replacement mapping is $\{(1,1),(2,2)\}$, and $g(x, y, z)$ will be replaced by `keywordSearch`(x, y). □

Similarly, a dataflow d is suitable as a subdataflow for f with signature ϑ, if we can map the free variables in $expr(d)$ to complex types in ϑ, such that the type assignment of d allows for application of the subtyping rule T.4.

Definition 3.14 (Replacement mapping). Let $\vartheta = \tau_1 \times \ldots \times \tau_n \rightarrow \tau_{out}$. Let e be an NRC expression, Γ a type assignment over $FV(e)$, and Θ a signature assignment over $SN(e)$. Let $\Gamma, \Theta \vdash e \colon \mu$.

We say that e, Γ and Θ are *replacement candidates* for ϑ, if and only if the following holds:

1. $\forall x \in FV(e) \, \exists i \in \{1, \ldots, n\} \colon \tau_i \preceq \Gamma(x)$, and
2. $\tau_{out} \preceq \mu$.

If e, Γ and Θ are replacement candidates for ϑ, then a *replacement mapping* for ϑ by e and Γ is a function $rep \colon FV(e) \rightarrow \{1, \ldots, n\}$, such that

$$\forall x \in FV(e) \colon \tau_{rep(x)} \preceq \Gamma(x) \, .$$

We are finally ready to define a binding tree.

Definition 3.15 (Binding tree). A binding tree for a dataflow d is a finite structure (T, λ, M), where

- $T = (r, V, E)$ is a finite tree with root r, $r \in V$ and $E \subseteq V \times V$;
- λ is a function defined on $V \cup E$, that labels each node with either a dataflow identifier or an external service identifier, and each edge with a service name;
- M is a function defined on $V \setminus \{r\}$;

such that the following properties are satisfied:

1. $\lambda(r) = d$.
2. Only leaves can be labeled with external service identifiers.
3. If node x is labelled with $d' \in \mathcal{D}$, then x has precisely as many children as there are service names in $SN(expr(d'))$, and for each $f \in SN(expr(d'))$, there is exactly one child edge labelled with f.
4. If node x is a leaf with parent y, where

- $\lambda(x) \in \mathcal{E}$
- $\lambda(y) = d' \in \mathcal{D}$, and
- the edge (y, x) is labelled with f,

then $sig(\lambda(x))$ is a signature replacement candidate for $signatures(d')(f)$. Moreover, $M(x)$ is a signature replacement mapping for $signatures(d')(f)$ by $sig(\lambda(x))$.

5. If node x is an internal node, and z is a child of x, where

   - $\lambda(x) = d' \in \mathcal{D}$,
   - $\lambda(z) = d'' \in \mathcal{D}$, and
   - $\lambda((x, z)) = f$,

   then $expr(d'')$, $types(d'')$, and $signatures(d'')$ are replacement candidates for $signatures(d')(f)$. Moreover, $M(x)$ is a replacement mapping for $signatures(d')(f)$ by $expr(d'')$ and $types(d'')$.

Observe that each subtree of a binding tree is again a binding tree (with λ and M restricted to this particular subtree).

Note 3.2.2. Let $d \in \mathcal{D}$ and $e = expr(d)$. We sketch a recursive procedure by which all possible binding trees (T, λ, M) for d can be constructed, with $T = (r, V, E)$.

If $SN(e) = \emptyset$, then

- $V = \{r\}$;
- $E = \emptyset$;
- $[\![\lambda]\!] = \{(r, d)\}$;
- $[\![M]\!] = \emptyset$.

If $SN(e) \neq \emptyset$, then let $\Gamma = types(d)$ and $\Theta = signatures(d)$.

Choose $I \subseteq SN(e)$, such that I contains all service names that you want to bind to subdataflows. Then $E = SN(e) \setminus I$ contains the service names that you want to bind to external services.

Construct a mapping $external^d$ from E to $\mathcal{E}$, such that for each $g \in E$, $sig(external^d(g))$ is a signature replacement candidate for $\Theta(g)$. Then, for each $g \in E$, construct a signature replacement mapping rep_g for $\Theta(g)$ by $sig(external^d(g))$.

Construct a mapping $internal^d$ from I to $\mathcal{D}$, such that for each $f \in I$, e_f, Γ_f and Θ_f are replacement candidates for $\Theta(f)$, with $e_f = expr(internal^d(f))$, $\Gamma_f = types(internal^d(f))$, and $\Theta_f = signatures(internal^d(f))$. Then, for each $f \in I$, construct a replacement mapping rep_f for $\Theta(f)$ by e_f and Γ_f.

Now, for each $f \in I$, recursively construct a binding tree (T_f, λ_f, M_f) for $internal^d(f)$, with $T_f = (r_f, V_f, E_f)$.

Finally:

- Let $G = \{x_g \mid g \in E\}$ be a set of new nodes. Then
$$V = \{r\} \cup G \cup \bigcup\{V_f \mid f \in I\}\,;$$
- $E = \{(r, x_g) \mid g \in E\} \cup \{(r, r_f) \mid f \in I\} \cup \bigcup\{E_f \mid f \in I\}$;
- $[\![\lambda]\!] = \{((r, x_g), g) \mid g \in E\} \cup \{((r, r_f), f) \mid f \in I\} \cup \bigcup\{[\![\lambda_f]\!] \mid f \in I\}$;
- $[\![M]\!] = \{(x_g, rep_g) \mid g \in E\} \cup \{(r_f, rep_f) \mid f \in I\} \cup \bigcup\{[\![M_f]\!] \mid f \in I\}$.

$\square$

Remember, that before executing an NRC expression we need a signature assignment and a function assignment for its service names. A binding tree β for a dataflow d encodes a signature assignment for $expr(d)$ that may differ from $signatures(d)$. Moreover, β associates service names in $expr(d)$ with services that can be actually executed. What we need now is a function assignment for $expr(d)$ that is consistent with the signature assignment encoded in β.

We define a function assignment specified by a binding tree by induction on the height of the tree.

Definition 3.16. Let $\beta = (T, \lambda, M)$ be a binding tree for d. Let $e = expr(d)$ and $\Theta = signatures(d)$.

We define a function assignment ζ_β for e as follows. Let h be the height of β.

If $h = 1$, then ζ_β is empty.

If $h > 1$, then, for $f \in SN(e)$, $\zeta_\beta(f)$ is defined as follows. Let x be the node such that $\lambda((r, x)) = f$.

- If $\lambda(x) = s \in \mathcal{E}$, then let $rep_s = M(x)$. Let $\Theta(f) = \tau_1 \times \ldots \times \tau_n \to \tau_{out}$ and $sig(s) = \mu_1 \times \ldots \times \mu_m \to \mu_{out}$. Then we define $\zeta_\beta(f)$ to be the following relation:
$$\{\, (v_1, \ldots, v_n, w) \mid \forall i \in \{1, \ldots, n\} \colon v_i \in [\![\tau_i]\!]\,, \\ \text{and } w = func(s)(v_{rep_s(1)}, \ldots, v_{rep_s(m)}) \,\}\,.$$

- If $\lambda(x) \in \mathcal{D}$, then let $e_f = expr(\lambda(x))$, and $rep_f = M(x)$. Let $\Theta(f) = \tau_1 \times \ldots \times \tau_n \rightarrow \tau_{out}$. Consider the subtree β_f of β rooted at x. By induction, we already have a function assignment ζ_{β_f} for e_f. Then we define $\zeta_\beta(f)$ to be the following relation:

$$\begin{aligned}\{\,(v_1, \ldots, v_n, w) \mid \forall i \in \{1, \ldots, n\}\colon v_i \in [\![\tau_i]\!]\,, \\ \sigma = [\,(y, v_{rep_f(y)}) \mid y \in FV(e_f)\,]\,, \\ \text{and } \sigma, \zeta_{\beta_f} \models e_f \Rightarrow w\,\}\,.\end{aligned}$$

Note that the order of elements in σ is irrelevant.

3.3 Dataflow repository

3.3.1 A formal model of a dataflow repository

We are now in position to give a formal definition of a dataflow repository, which gathers all previously defined concepts. We use the following notations:

notation	the set of all possible
$\mathcal{S}$	*signatures*
$\mathcal{F}$	*service functions*
$\mathcal{NRC}$	*NRC expressions*
$\mathcal{TA}$	*type assignments*
$\mathcal{SA}$	*signature assignments*
$\mathcal{VA}$	*value assignments*
$\mathcal{B}indings$	*binding trees*
$\mathcal{R}uns$	*runs*
$\mathcal{T}riples$	*triples occurring in runs*

Definition 3.17 (Dataflow repository).

A dataflow repository contains the following pairwise disjoint sets:

- a finite set $\mathcal{E}$ of *external service identifiers*,
- a finite set $\mathcal{D}$ of *dataflow identifiers*, and
- a finite set $\mathcal{R}$ of *run identifiers*,

together with the mappings shown in the following list:

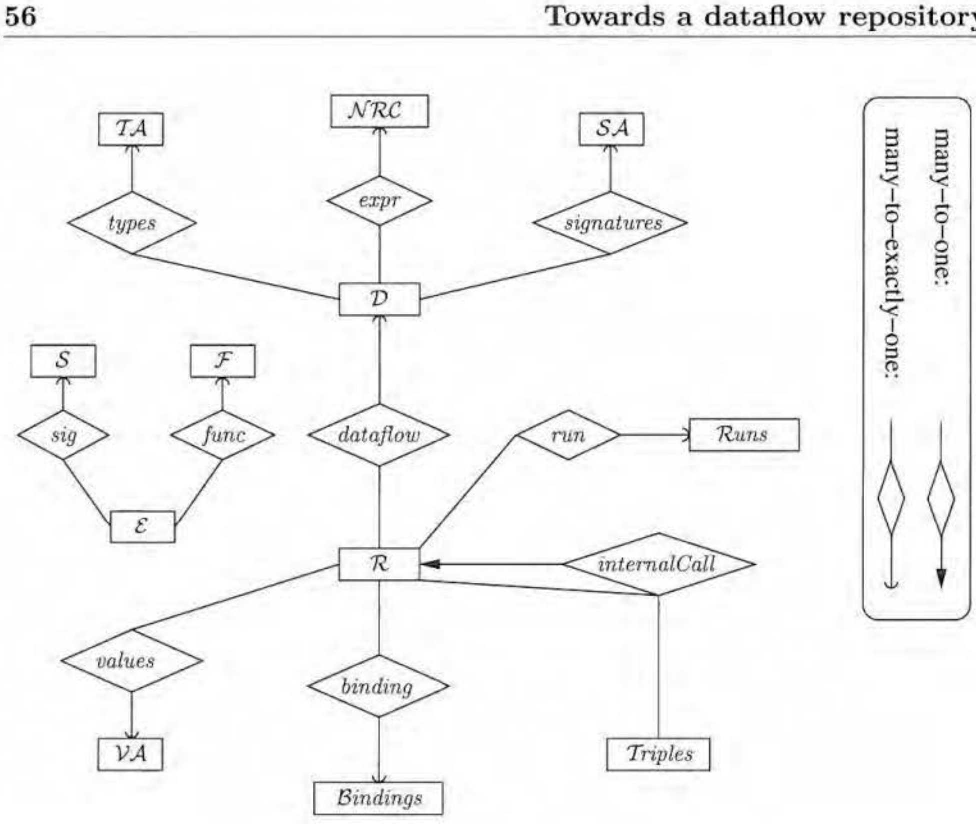


Figure 3.1: ER-diagram of a dataflow repository

$$\begin{aligned}
sig &: \mathcal{E} \to \mathcal{S} \\
func &: \mathcal{E} \to \mathcal{F} \\
expr &: \mathcal{D} \to \mathcal{NRC} \\
types &: \mathcal{D} \to \mathcal{TA} \\
signatures &: \mathcal{D} \to \mathcal{SA} \\
dataflow &: \mathcal{R} \to \mathcal{D} \\
values &: \mathcal{R} \to \mathcal{VA} \\
binding &: \mathcal{R} \to \mathcal{B}indings \\
run &: \mathcal{R} \to \mathcal{R}uns \\
internalCall &: \mathcal{R} \times Triples \to \mathcal{R}
\end{aligned}$$

The first nine mappings are standard, total, many-to-one mappings. The last mapping, however, is partial, but must be one-to-one. Figure 3.1 shows an ER-diagram of the used sets and mappings.

Moreover, the mappings must satisfy the following integrity constraints, for any $e \in \mathcal{E}$, any $d \in \mathcal{D}$ and any $r \in \mathcal{R}$:

- $sig(e)$ is *consistent* with $func(e)$;

- *types*(d) is defined on $FV(\mathit{expr}(d))$;
- *signatures*(d) is defined on $SN(\mathit{expr}(d))$;
- *expr*(d) is well-typed under *types*(d) and *signatures*(d);
- *values*(r) is defined on $FV(\mathit{expr}(\mathit{dataflow}(r)))$;
- *values*(r) is *consistent* with $\mathit{types}(\mathit{dataflow}(r))$.
- The root of *binding*(r) is labeled with *dataflow*(r).
- *run*(r) is a run of $\mathit{expr}(\mathit{dataflow}(r))$ evaluated under *values*(r) and $\zeta_{\mathit{binding}(r)}$, i.e., $\mathit{values}(r), \zeta_{\mathit{binding}(r)} \approx\!\!\!\models \mathit{expr}(\mathit{dataflow}(r)) \Rightarrow \mathit{run}(r)$.
- The repository is *closed* by the mapping *internalCall*.

We still need to define the last constraint in the above definition. When a dataflow *dataflow*(r) is being executed to construct a run for r, then for each occurrence of a service-call expression f in $\mathit{expr}(\mathit{dataflow}(r))$, if f is bound to a subdataflow, the service function $\zeta_{\mathit{binding}(r)}(f)$ must be used to construct the run-triple t for f. However, in a dataflow repository, $\zeta_{\mathit{binding}(r)}(f)$ does not really exist. Instead, the subdataflow bound to f will be executed, the resulting run, identified by r', will be stored, and the final result value of $\mathit{run}(r')$ will be used to construct t. Moreover, the pair $((r,t),r')$ must be added to *internalCall*. If f occurs inside a for-loop, then its subdataflow may be executed several times.

The last constraint, *closure*, corresponds to the following intuition: if the repository contains a run of some dataflow, then it also contains all corresponding runs of its subdataflows. This is precisely the purpose of the mapping *internalCall*. Formally, we define:

Definition 3.18. A dataflow repository is *closed* by *internalCall*, if for any $r \in \mathcal{R}$ and any $t = (\Phi, \sigma, v) \in \mathit{Triples}$, the following holds:

- $\mathit{internalCall}(r,t)$ is defined if and only if $t \in \mathit{run}(r)$ and Φ is an occurrence of a service-call expression to a service named f, such that the node x in *binding*(r), which is connected to the root by the edge labeled with f, has a dataflow ID as label, say d'.
- If $\mathit{internalCall}(r,t)$ is indeed defined, say $\mathit{internalCall}(r,t) = r'$, then
  - $\mathit{dataflow}(r') = d'$.
  - $\mathit{binding}(r')$ equals the subtree of $\mathit{binding}(r)$ rooted at x.

– let the service call be of the form $f(e_1, \ldots, e_n)$ and $binding(r) = (T, \lambda, M)$. So there are n run-triples of the form $(\Phi \cdot [i, e_i], \sigma, v_i)$ in $run(r)$, for $i \in \{1, \ldots, n\}$. Then $values(r')$ is equal to σ extended with $[(y, v_{M(x)(y)}) \mid y \in FV(expr(d'))]$.
– the final result value of $run(r')$ equals v.

What we have not included in our formal model are annotations (meta-data). However, it is possible to extend the model, for instance, by adding relations from annotation identifiers (representing diverse meta-data) to identifiers and entities defined in the model. The actual content of meta-data is an ongoing research topic [DCM, MDF+10, LKM+09], beyond the scope of this work.

3.3.2 Proof-of-concept representation in SQL:2003

Complex data

Complex data flowing in a scientific workflow can be either atomic in a workflow, or can have a structure that must be available to the operations in the workflow. We model complex data by complex values, defined in Section 2.1.

Remember that "atomic" data can be quite complex, e.g., a text file or an XML document. However, for a dataflow that has only tasks that operate on the file as a whole, it is not relevant to model the file contents. In such cases, we model the file as a base value. On the other hand, if the structure of the file as a collection is important, because we want to apply some operation to its elements, then we model the file as a complex value, e.g., as a set of records.

We first discuss the storing of base values. Then we discuss two basic ways for storing complex values: *decomposition* and *XML representation*.

Base Values. We can represent most base values as `VARCHAR` strings. For small types of atomic data, such as numbers, words, short texts or dates, the `VARCHAR` string can hold the entire string representation of the value. Such values can be easily stored together with an entity they are part of, such as a complex value, a value assignment or a run triple.

For large atomic data such as a file, we can also represent it as a `VARCHAR` string by means of an identifier of the file (e.g., full path name or an URI). In many cases, however, it is more desirable to store large atomic data in the database as a `BLOB` (which can contain a text file, or an XML document, as well as a binary file) in a table `BaseValues(vID VARCHAR, object BLOB)`. In that case, the `VARCHAR` string representing the data object is an identifier, and a foreign key to `BaseValues(vID)`.

Decomposition of complex values. A complex value, together with its nested subvalues, can be naturally viewed as a tree (Section 2.1.2). We generate a `VARCHAR` string ID for each tuple and set node. The base values, which occur as leaves in the tree, already have their `VARCHAR` string representations. We then store the complex-value tree in two tables:

`Sets(vID VARCHAR, kind CHAR(1), eValID VARCHAR)`, and
`Tuples(vID VARCHAR, kind CHAR(1), lbl VARCHAR, cValID VARCHAR)`.

Here, `eValID` stands for element ID, `lbl` stands for label, and `cValID` stands for component ID. Attribute `kind` specifies the kind of values in attributes `eValID` and `cValID` as follows:

- value `b` for a base value fully represented by the `VARCHAR` string,
- value `B` for an ID of a base value stored in the `BaseValues` table,
- value `s` for an ID of a set value stored in the `Sets` table,
- value `t` for an ID of a tuple value stored in the `Tuples` table, or
- value `e` in the record `('emptyset', 'e', NULL)` in table `Sets`, which represents the empty set.

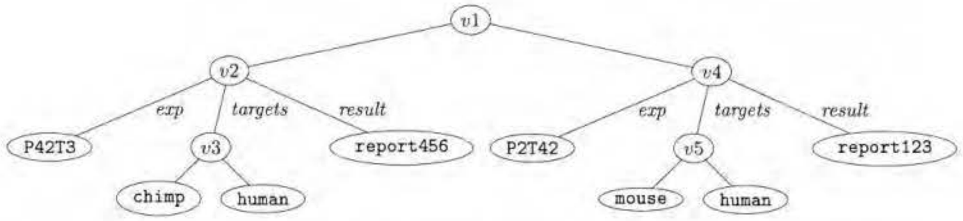
Database constraints must enforce the existence of referenced complex-value identifiers in their corresponding tables.

Example 3.19. Consider the following complex value:

$$\{\langle exp: \texttt{P2T42}, targets: \{\texttt{human}, \texttt{mouse}\}, result: \texttt{report123}\rangle,$$
$$\langle exp: \texttt{P42T3}, targets: \{\texttt{human}, \texttt{chimp}\}, result: \texttt{report456}\rangle\}.$$

Figure 3.2 shows its tree representation and decomposition. □

XML representation of complex values. We can also take advantage of the XML data type supported by modern database systems. We can store a complex-value tree as an XML value either in table `Values(vID VARCHAR, value XML)`, or together with an entity the complex value is a part of, like a value assignment or a run triple. Base values at the leaves can be represented by text nodes. For a large base value, the text node contains its ID in the `BaseValues` table. The DTD for complex values is in Section A.3.1. If the complex value is stored in `Values`, we assume that the `valID` attribute of the



Sets		
vID	kind	eValID
v1	t	v2
v1	t	v4
v3	b	chimp
v3	b	human
v5	b	human
v5	b	mouse

Tuples			
vID	kind	lbl	cValID
v2	b	exp	P42T3
v2	B	result	report456
v2	s	targets	v3
v4	B	result	report123
v4	b	exp	P2T42
v4	s	targets	v5

Figure 3.2: Tree representation and decomposition of a complex value

root element of the value stored in `Values(value)` matches the corresponding value in `Values(vID)`. Figure 3.3 shows the XML representation of the complex value from Example 3.19.

There is an additional choice when a complex value contains XML documents at the leaves: we can just have the IDs of these base values in the `BaseValues` table, or we can include their full XML content. For example, in Figure 3.3, the results are represented by IDs `report123` and `report456` referring to the `BaseValues` table, but alternatively we could have replaced these IDs inside the XML tree by the corresponding full XML values.

The best choice among decomposition, intermediate XML, and full XML for complex values depends on the application. Library routines can be provided for conversion between the representations; these routines can then be used in SQL statements.

Complex types

We store complex types in the following tables:

`BaseTypes(tID VARCHAR)`,
`SetTypes(tID VARCHAR, kind CHAR(1), etID VARCHAR)`, and
`TupleTypes(tID VARCHAR, kind CHAR(1), lbl VARCHAR, ctID VARCHAR)`.

Here, `etID` stands for element type ID, `lbl` stands for label, and `ctID` stands for component type ID. Attribute `kind` specifies the kind of types in attributes

```
<set valID="v1">
  <tuple valID="v4">
    <lbl>exp</lbl>
    <base kind="b">P2T42</base>
    <lbl>targets</lbl>
    <set valID="v5">
      <base kind="b">human</base>
      <base kind="b">mouse</base>
    </set>
    <lbl>result</lbl>
    <base kind="B">report123</base>
  </tuple>
  <tuple valID="v2">
    <lbl>exp</lbl>
    <base kind="b">P42T3</base>
    <lbl>targets</lbl>
    <set valID="v3">
      <base kind="b">human</base>
      <base kind="b">chimp</base>
    </set>
    <lbl>result</lbl>
    <base kind="B">report456</base>
  </tuple>
</set>
```

Figure 3.3: XML representation of a complex object

```
<settype typeID="Experiments">
  <tupletype typeID="Experiment">
    <lbl>exp</lbl>
    <basetype typeID="ExpNr"/>
    <lbl>targets</lbl>
    <settype typeID="Organisms">
      <basetype typeID="Organism"/>
    </settype>
    <lbl>result</lbl>
    <basetype typeID="Report"/>
  </tupletype>
</settype>
```

Figure 3.4: XML representation of a complex type

etID and ctID as follows:

- value b for an ID of a base type stored in the BaseTypes table,
- value s for an ID of a set type stored in the SetTypes table, or
- value t for an ID of a tuple type stored in the TupleTypes table.

Database constraints must enforce the existence of referenced complex-type identifiers in their corresponding tables. We assume that type bottom is stored in BaseTypes.

We also need an XML representation for complex types, as complex types also feature as parts of other entities, like a type assignment or a signature assignment. The DTD for complex types is in Section A.3.2. Figure 3.4 shows the XML representation of a possible complex type for value from Example 3.19.

```
dataflow AFlow(input: aSetType)
  for x in input return f(x)


dataflow BFlow(input: aTupleType)
  <c: f(g(input.a)), d: f(g(input.b))>
```

Figure 3.5: Dataflow specifications

```
<expr eID="BFlow">
  <tupleExpr eID="e1">
    <lbl>c</lbl>
    <call eID="e2">
      <service>f</service>
      <call eID="e3">
        <service>g</service>
        <project eID="e4">
          <var eID="e5">input</var>
          <lbl>a</lbl>
        </project>
      </call>
    </call>
    <lbl>d</lbl>
    <call eID="e6">
      <service>f</service>
      <call eID="e7">
        <service>g</service>
        <project eID="e8">
          <var eID="e9">input</var>
          <lbl>b</lbl>
        </project>
      </call>
    </call>
  </tupleExpr>
</expr>
```

Figure 3.6: XML representation of BFlow

NRC dataflows

Figure 3.5 shows example specifications of dataflows. A dataflow consists of

- a keyword `dataflow` followed by a name, which may be used as the dataflow's identifier;
- a type assignment for the free variables of the dataflow expression, between parentheses; and
- an NRC expression, which specifies the dataflow's behaviour.

A dataflow specification is stored in two tables:

Dataflows(dID VARCHAR, expr XML), and
Variables(dID VARCHAR, var VARCHAR, type VARCHAR, kind CHAR(1)).

In table Dataflows, attribute dID contains the identifier of the dataflow, and attribute expr contains the NRC expression, in XML format based on its BNF syntax. Table Variables stores the type assignment of the dataflow. Attribute dID is a foreign key to Dataflows(dID). Attribute kind uses again values b, s and t to specify the kind of the complex type.

Recall that we also need a signature assignment for the service names occurring in the dataflow. In line with common usage in programming languages, we use parameter names to refer to positions in a signature. The signature assignment is stored in table

```
Signatures(dID VARCHAR, service VARCHAR, par VARCHAR,
                                    type VARCHAR, kind CHAR(1)),
```

with dID again a foreign key to Dataflows(dID), and kind having the same function as in table Variables. Attribute par contains a parameter name. We asume that parameter name output is reserved for the output type of the signature. For tables Variables an Signatures, database constraints must enforce the existence of referenced complex-type identifiers in their corresponding tables.

To facilitate storing dataflows, the system should provide a stored function

```
InsDesign(expr XML, types XML, sigs XML) returns VARCHAR,
```

which expects, respectively, an NRC expression, a type assignment, and a signature assignment, all in their corresponding XML format.

Function InsDesign should ensure that (i) a dataflow specification is not stored in the repository unless it is well-typed under the given type assignment and signature assignment; (ii) if the eID of the root element of parameter expr already exists as another dataflow's identifier, a new dataflow ID is generated; (iii) upon successful completion, value of attribute Dataflows(dID) of the new record matches the returned dataflow ID; and (iv) in the value of Dataflows(expr), the eID attribute of the root element also contains the dataflow ID, and others have an eID composed of e followed by a number in document order, starting at one.

Note that instead of using tables Variables and Signatures for storing, respectively, the type assignment and the signature assignment of a dataflow, we can also add attributes of type XML to table Dataflows, and store their XML representations.

The DTDs for the type and signature assignments are in, respectively, Sections A.3.3 and A.3.4. The DTD for NRC expressions is in Section A.3.5, and

Figure 3.6 shows the XML representation for dataflow Bflow. Note that all expression element nodes in the XML tree have unique eID attributes. This allows us to create an index on XML column expr based on the XPath pattern //@eID. This is useful to support efficient querying of stored expressions using SQL/XML. Indeed, other tables in the repository contain references to these IDs, so queries may use conditions involving the above XPath pattern.

External services

In order to integrate external services in the dataflow repository, we assume Java wrapper functions for them, which are registered as external routines (user-defined Java functions). These functions take XML representations of complex objects as input and output. In this way, external services can be called directly in SQL statements, but also, dataflow executions can be initiated from inside the database server.

These Java wrapper functions must be registered in table

External(<u>extID</u> VARCHAR, name VARCHAR, checks VARCHAR),

where attribute extID is an external service identifier that can be used in a binding tree, and name is the name of the Java wrapper registered in the system's catalogue. Their signatures are stored in table

ExtSigs(<u>extID</u> VARCHAR, <u>par</u> VARCHAR, type VARCHAR, kind CHAR(1)),

with extID a foreign key to External(extID), and kind having the same function as in table Variables. Attribute par contains a parameter name. We asume that parameter name output is reserved for the output type of the signature, and that database constraints enforce the existence of referenced complex-type identifiers in their corresponding tables.

The designer of the Java wrapper function has the responsibility to ensure that it only executes its external service if the XML values supplied for its parameters match the registered complex type. Therefore, we assume that each Java wrapper function provides an associated user-defined function, of the same signature except for the output type, that can be used to verify if given XML values conform to the registered signature. The output type should allow for necessary communication about possible errors. The name of this function must be stored in External(checks). This function can then be used, in case of failure, to rule out type mismatches.

```
<btree dID="AFlow">
 <subentry>
   <service>f</service> <sub>BFlow</sub>
   <subpair>
     <var>input</var> <spar>x</spar>
   </subpair>
   <btree dID="BFlow">
     <extentry>
       <service>f</service> <ext>funcA</ext>
       <extpair>
         <epar>a</epar> <spar>x</spar>
       </extpair>
     </extentry>
     <extentry>
       <service>g</service> <ext>funcB</ext>
       <extpair>
         <epar>b</epar> <spar>x</spar>
       </extpair>
     </extentry>
   </btree>
 </subentry>
</btree>
```

Figure 3.7: A binding tree for AFlow

Executing and storing runs

Remember that, in order to execute a dataflow, we must have a value assignment and a function assignment for the dataflow. A value assignment is represented by XML format defined in the DTD in Section A.3.6. A function assignment is provided indirectly by a binding tree, that associates service names used in the dataflow to either subdataflows, or external services. A binding tree is represented by XML format defined in the DTD in Section A.3.7.

Example 3.20. Recall dataflow `AFlow` from Figure 3.5. To execute `AFlow` we might want to bind service `f(x: aTupleType)` with output type `another-TupleType` to `BFlow(input: aTupleType)`, so `BFlow` becomes a subdataflow of `AFlow`. We now bind the service names in `BFlow`, `f(x: Int)` and `g(x: Int)`, both with output type `Int`, to some external functions `funcA(a: Int)` and `funcB(b: Int)`, respectively, both with output type `Int`. The binding tree that specifies all this is shown in Figure 3.7. □

To execute a dataflow stored in the dataflow repository, the system should provide a stored procedure

```
Execute(dID: VARCHAR, vassign: XML, btree: XML),
```

where `dID` designates a dataflow specification from the `Dataflows` table, `btree` is a binding tree for the dataflow, and `vassign` is an assignment of input values to the free variables of the dataflow. It is the task of `Execute` to ensure the consistency between the type and value assignments of the dataflow, the validity of the binding tree, and the consistency between the signature and function assignments that can be derived from the given binding tree. It must also determine the output type of the dataflow for this run.

A successful call to `Execute` should result in the run of the dataflow being stored in two tables:

```
Runs(rID VARCHAR, dID VARCHAR, vassign XML, btree XML,
                                    type VARCHAR, kind CHAR(1)),
Triples(rID VARCHAR, caller VARCHAR, cvassign XML,
                          subexpr VARCHAR, vassign XML, value XML).
```

Table `Runs` stores the environment under which the run was executed (value assignment and binding tree), with attribute `rID` a newly generated run identifier, and `dID` a foreign key to `Dataflows(dID)`. Attributes `type` and `kind` hold the complex type of the final result of the run.

Table `Triples` contains "tagged" triples for the run: one holding the final result value, and one triple for each service call that has been made.[||] The system should provide a stored function `Run(rID: VARCHAR)`, that can later reconstruct all other triples. The triples can be either returned in some format, or `Run` can return the name of a temporary table storing the triples.

The attributes of table `Triples` have the following meaning. Consider the service-call expressions in the NRC expression of the main dataflow. Attribute `subexpr` holds the `eID` of the corresponding element in the XML representation of the dataflow expression. Attribute `vassign` holds the value assignment of the free variables of the dataflow at the time of the call. Attribute `value` holds the final result value of the service call. For the tuple holding the final result value of the run, `subexpr` is simply the `eID` identifier of the first child of the root element, and `vassign` is the original value assignment. Attributes `caller` and `cvassign` hold `NULL` for all the triples corresponding to the execution of the main dataflow. In triples corresponding to subdataflow executions, these columns hold, respectively, the `eID` of the calling subexpression, and the value assignment of the dataflow parameters at the time of the call. Other attributes contain the same values as if the subdataflow were executed as a

[||]In some cases it may be better to store additional triples in table `Triples`. The system may therefore provide means to alter the behaviour of `Execute`.

main dataflow.[**]

Example 3.21. Let us illustrate the `Triples` table. Assume we have executed `BFlow` from Figure 3.5 on value[††] $\langle a\colon 2, b\colon 6\rangle$, for the binding tree that can be found as the subtree in Figure 3.7. A possible run with ID `run1` could generate the following triples:

`('run1', NULL, NULL, 'e1',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$, $\langle c\colon 1, d\colon 0\rangle$`),`
`('run1', NULL, NULL, 'e2',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 1),`
`('run1', NULL, NULL, 'e3',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 4),`
`('run1', NULL, NULL, 'e4',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 2),`
`('run1', NULL, NULL, 'e5',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$, $\langle a\colon 2, b\colon 6\rangle$`),`
`('run1', NULL, NULL, 'e6',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 0),`
`('run1', NULL, NULL, 'e7',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 1),`
`('run1', NULL, NULL, 'e8',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 6),`
`('run1', NULL, NULL, 'e9',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$, $\langle a\colon 2, b\colon 6\rangle$`),`

of which only the following would be stored in table `Triples` (`e2` and `e6` are calls to `f`, and `e3` and `e4` are calls to `g`):

`('run1', NULL, NULL, 'e1',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$, $\langle c\colon 1, d\colon 0\rangle$`),`
`('run1', NULL, NULL, 'e2',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 1),`
`('run1', NULL, NULL, 'e3',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 4),`
`('run1', NULL, NULL, 'e6',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 0),`
`('run1', NULL, NULL, 'e7',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 1).`

Now assume we have executed `AFlow` for the binding tree from Figure 3.7, and on value *in* equal $\{\langle a\colon 2, b\colon 6\rangle, \langle a\colon 5, b\colon 35\rangle\}$. From a possible run with ID `run2`, the following triples would be stored in table `Triples` (`e4` is a call to `f`, which is bound to `BFlow`):

`('run2', NULL, NULL, 'e1',` $[(\mathit{input}, \mathit{in})]$, $\langle c\colon 1, d\colon 0\rangle$`),`
`('run2', NULL, NULL, 'e4',` $[(\mathit{input}, \mathit{in}), (x, \langle a\colon 2, b\colon 6\rangle)]$, $\langle c\colon 1, d\colon 0\rangle$`),`
`('run2', NULL, NULL, 'e4',` $[(\mathit{input}, \mathit{in}), (x, \langle a\colon 5, b\colon 35\rangle)]$, $\langle c\colon 1, d\colon 0\rangle$`),`
`('run2', 'e4',` $\mathit{cassing}_1$`, 'e1',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$, $\langle c\colon 1, d\colon 0\rangle$`),`
`('run2', 'e4',` $\mathit{cassing}_1$`, 'e2',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 1),`
`('run2', 'e4',` $\mathit{cassing}_1$`, 'e3',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 4),`
`('run2', 'e4',` $\mathit{cassing}_1$`, 'e6',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 0),`
`('run2', 'e4',` $\mathit{cassing}_1$`, 'e7',` $[(\mathit{input}, \langle a\colon 2, b\colon 6\rangle)]$`, 1),`
`('run2', 'e4',` $\mathit{cassing}_2$`, 'e1',` $[(\mathit{input}, \langle a\colon 5, b\colon 35\rangle)]$, $\langle c\colon 1, d\colon 0\rangle$`),`
`('run2', 'e4',` $\mathit{cassing}_2$`, 'e2',` $[(\mathit{input}, \langle a\colon 5, b\colon 35\rangle)]$`, 1),`
`('run2', 'e4',` $\mathit{cassing}_2$`, 'e3',` $[(\mathit{input}, \langle a\colon 5, b\colon 35\rangle)]$`, 4),`
`('run2', 'e4',` $\mathit{cassing}_2$`, 'e6',` $[(\mathit{input}, \langle a\colon 5, b\colon 35\rangle)]$`, 0),`
`('run2', 'e4',` $\mathit{cassing}_2$`, 'e7',` $[(\mathit{input}, \langle a\colon 5, b\colon 35\rangle)]$`, 0),`

[**]In fact, this is a representation of the triples of a run together with the mapping *internalCall*.

[††]We use the formal representation of value assignments and complex values instead of their XML-representations, for brevity.

with

$$\begin{aligned} cassing_1 &= [(input, in), (x, \langle a\colon 2, b\colon 6\rangle)]\,, \\ cassing_2 &= [(input, in), (x, \langle a\colon 5, b\colon 35\rangle)]\,. \end{aligned}$$

Note that $\{(2,4),(5,4),(6,1),(35,0)\} \subseteq [\![\texttt{funcB}]\!]$, which is bound to g, and $\{(4,1),(1,0),(0,0)\} \subseteq [\![\texttt{funcA}]\!]$, which is bound to f. □

Procedure Execute can be straightforwardly implemented by compiling NRC into SQL/XML. Indeed, under a decomposed representation of complex objects, NRC operations can be quite simply programmed in SQL. We have already seen that external services can be called in SQL as external routines. Under the XML representation of complex objects, either decomposition can be applied first (this is the approach we have tested), or a direct compilation of the NRC operations into XQuery may be performed.

Annotations

Another advantage of implementing a dataflow repository on top of a modern SQL platform, is the possible integration with an annotation system, if it also provides a database schema. The identifiers used in the dataflow repository can be referenced as foreign keys in annotation tables.

The are kinds of annotations though, that should be provided by the repository system. These annotations are typically starting and ending times of runs, the user performing the run, and properties of used external services (e.g., version number, starting and ending time of the call, possible errors codes). In order to provide applications with a flexible annotation recording of runs, procedure Execute can provide hooks that are executed before and after each service call, and at the beginning and at the end of its own execution. The application developer can instantiate these hooks with the code required to record the meta-data required by the application. In a similar way, function InsDesign can provide hooks to record annotations such as the author of the dataflow, and the time of its addition to the repository.

Choice of representation Note that this is only one of the possible representations of the formal model. It is impossible to provide one representation that is equally suitable for all scientific workflow applications, as there are many issues to consider, e.g., the size, complexity, and volume of data; the number of different dataflows and how often they are executed.

We have chosen this representation because it closely resembles the formal model. Even if the actual database schema is different, it is possible to create (materialised) views that simulate this representation.

4

Querying a dataflow repository

4.1 Querying an SQL/XML dataflow repository

The participants of the Provenance Challenges* have already informally formulated various queries, involving both a dataflow specification and its past executions.

For example, for a specified part of a workflow output, say *out*, they have formulated queries that ask (i) which workflow inputs have contributed to the computation of *out* (Q1,Q5 from PC3); (ii) which part of the execution contributed to the computation of *out*, possibly further restricted by annotations, or only up to a specified task (Q1-Q3 from PC1, Q3 from PC3); (iii) to verify if certain tasks were involved in the computation of *out* (Q2 of PC3); (iv) to look for tasks that can be swapped during execution without affecting *out* (optional Q5 from PC3).

Queries that involve many executions of the same workflow ask (i) to find all invocations of a specified task, using a specified input, and having specified annotations (Q4); (ii) to retrieve (intermediate) results produced by a specified task and/or having specified annotations (Q8-Q9 from PC1), or even preceded by another specified task (Q6 from PC1); (iii) to find all workflow outputs produced from a specified input (Q5 from PC1); (iv) to find differences between specified past executions (Q7 from PC1).

In general, there are various types of queries that a dataflow repository should support, including:

*`http://twiki.ipaw.info/bin/view/Challenge/WebHome`, we refer to the first challenge as PC1, and to the third as PC3.

- Queries involving *subvalues* of a (final) result. Indeed, in some dataflows, both intermediate values and the final result value may be huge data sets, and the user might be only interested in some part.
- Querying vast amounts of past executions, in order to identify dataflows and their executions involving a particular external service. Indeed, if that service produced erroneous results, or there is a better implementation available, such queries are necessary if we want to rerun the affected dataflows with another external service.
- Queries that allow modifying of dataflow specifications and immediate execution of the modified dataflows.

In this section, we give some example queries in SQL/XML, based on the proof-of-concept schema from Section 3.3.2.

Queries involving subvalues. Recall dataflow `Bflow` from Figure 3.5, and the run with ID `run1` from Example 3.21, with final result value $\langle c\colon 1, d\colon 0\rangle$. Consider now the query:

> "What is the part of the run that produced the subvalue 1 in the output?"

From the dataflow specification we see that subvalue 1 is the output of `f`, applied to the output of `g`, applied to `input.a`. We can thus retrieve the three relevant triples as follows:

```
SELECT 'input.a', T.vassign, T.value
  FROM Triples T, Dataflows D
 WHERE T.rID='run1' AND D.dID='Bflow'
   AND XMLEXISTS('$e//project[@eID=$s and ./lbl="a"]'
                 PASSING D.expr AS "e", T.subexpr AS "s")
 UNION
SELECT 'g', T.vassign, T.value
  FROM Triples T, Dataflows D
 WHERE T.rID='run1' AND D.dID='Bflow'
   AND XMLEXISTS('$e//call[@eID=$s and ./service="g"
                      and ./project/lbl="a"]'
                 PASSING D.expr AS "e", T.subexpr AS "s")
 UNION
SELECT 'f', T.vassign, T.value
  FROM Triples T, Dataflows D
 WHERE T.rID='run1' AND D.dID='Bflow'
```

```
  AND XMLEXISTS('$e//call[@eID=$s and ./service="f"
                   and ./call/project/lbl="a"]'
                PASSING D.expr AS "e", T.subexpr AS "s")
```

Note the use of the SQL/XML predicate XMLEXISTS [EM04, ÖCKM06] to retrieve the `eID` attributes of the elements in the XML representation of the dataflow's NRC expression.

Things get a bit more subtle when working with collections and subdataflows.

Recall dataflow `Aflow` from Figure 3.5, and the run with ID `run2` from Example 3.21, with final result value $\{\langle c\colon 1, d\colon 0\rangle\}$. Suppose we again want to know the part of the run that produced the subvalue 1. From the dataflow specification we see that subvalue 1 is part of the result of `f` applied to at least one element of the input collection. We thus want to retrieve all elements `x` in `input` for which `f(x)` contains subvalue 1:

```
SELECT 'f', XMLQUERY('$b//varval[./var="x"]' PASSING T.vassign AS "b")
  FROM Triples T, Dataflows D
 WHERE T.rID='run2' AND D.dID='Aflow'
  AND XMLEXISTS('$e//call[@eID=$s and ./service="f"]'
                PASSING D.expr AS "e", T.subexpr AS "s")
  AND XMLEXISTS('$v[./lbl[1]="c" and ./base[2]="1"
                   and ./lbl[3]="d" and ./base[4]="0"]'
                PASSING T.value AS "v")
```

Further querying must be done in the triples corresponding to the run of `BFlow`, executed as a subdataflow of `Aflow`, in the same way as in the first example. Note the use of the SQL/XML function XMLQUERY to extract the value of variable `x` from the value assignment of the triple.

The two example queries over `run1` and `run2` are simple but typical examples of provenance queries, where we ask for the part of the run that has contributed to a given value v, occurring as a subvalue of the output, or some intermediate result. In the context of workflows, "contribution" is often only informally understood, and may involve subvalues from the input that also occur in v, unchanged, as well as subvalues from the input that were transformed into v by a computation.

When the dataflow specification is known in advance, as in our examples, we have seen that provenance can be directly expressed in SQL/XML. This is no longer straightforward, however, when the dataflow specification is not fixed in the query, as it is the case for queries involving numerous runs of dataflows with different specifications.

Our solution is to provide a generic *subvalue provenance* computation as a library routine. In Sections 4.2–4.3 we formally specify which part of a run has contributed to a given subvalue, be means of inference rules that track the subvalue provenance.

Queries involving (external) services. For some queries, even involving numerous executions of different dataflows, we do not need to know the exact dataflow specification to pose the query. For example, queries like

> "List all bioinformatics dataflows in which a service named `f`, called with input `5`, returned the value `GPZ158`."

can be expressed using similar techniques as above (assuming an annotation table `BioInf(dID ...)` that lists the IDs of bioinformatics dataflows):

```
SELECT dID
  FROM Triples T1, Triples T2, Runs R, Dataflows D, BioInf B
 WHERE R.dID = D.dID AND D.dID = B.dID
   AND T1.rID = R.rID AND T2.rID = R.rID AND T1.vassign=T2.vassign
   AND XMLEXISTS('$e//call[@eID=$s1 and ./service="f"]/*[2][@ID=$s2]'
                 PASSING D.expr AS "e",
                 T1.subexpr AS "s1", T2.subexpr AS "s2")
   AND XMLEXISTS('$v="GPZ158"' PASSING T1.value AS "v")
   AND XMLEXISTS('$v="5"' PASSING T2.value AS "v")
```

Consider an external service that is registered in the database with ID `BLASTv1`. The database may contain many dataflow executions that have called this service. To retrieve them, it suffices to look in the binding tree of each execution, which is stored together with the run ID in the `Runs` table. The following query also retrieves the service name that is bound to `BLASTv1`.

```
CREATE VIEW Bv1Calls AS
  SELECT R.rID, Tree.service
    FROM Runs R, XMLTABLE('$tr//extentry' PASSING R.btree AS "tr"
                      COLUMNS service VARCHAR(30),
                              ext VARCHAR(30)) AS Tree
   WHERE Tree.ext='BLASTv1'
```

Now suppose we want to understand the effect of replacing the external function `BLASTv1` by another one, say, `BLASTv2`. We are interested, across all executions in the database, which calls to `BLASTv1` would give a different result when replaced by a call to `BLASTv2`. We can find this out using the following query:

```
SELECT O.rID, O.subexpr, O.argval, O.value, N.newvalue
  FROM (SELECT R.rID, U.subexpr, T1.value, T2.value AS argval
        FROM Runs R, Bv1Calls B, Dataflows D, Triples T1, Triples T2
       WHERE R.rID=B.rID AND R.dID=D.dID
         AND R.rID=T1.rID AND R.rID=T2.rID AND T1.vassign=T2.vassign
         AND XMLEXISTS
               ('$e//call[@ID=$s1 and ./service=$b]/*[2][@ID=$s2]'
                PASSING D.expr AS "e", T1.subexpr AS "s1",
                B.service AS "b",  T2.subexpr AS "s2")
     ) AS O,
     LATERAL (VALUES BLASTv2(O.argval)) AS N(newvalue)
WHERE is_different(O.value,N.newvalue)
```

Observe how the query directly calls BLASTv2 on the inputs of the recorded calls to BLASTv1. We also use a Boolean user-defined function is_different to compare the two resulting XML values, as a literal non-equality is not what we want.

Queries executing modified dataflow specifications. What if we want to find those dataflow executions whose *final result value* would change if we replaced BLASTv1 by BLASTv2? Note that a difference in an individual call might not result in a difference in the final result. To answer this query, we can no longer simply call BLASTv2 as before, because the call is embedded within an entire NRC expression, which is not fixed by the query, and which has to be re-evaluated.

(Of course, if we are only interested in the executions of a dataflow whose specification is known in advance, we can simply rerun it, either through the repository or directly in a query, and compare the differences.)

Our solution, inspired by earlier work on meta-querying [VGV96, VVV05, VVV04], lies in providing dynamic dataflow execution through a library function. More specifically, the system may provide a user-defined table-valued function Eval with the following signature:

```
function Eval(expr XML, vassign XML, btree XML)
  returns table (caller XML, cassign XML, subexpr integer,
                 vassign XML, value XML)
```

This function should return the set of tagged triples representing the whole run of NRC expression expr on value assignment vassign and binding tree btree. Essentially, Eval should be a lightweight version of procedure Execute, where the run is not stored in the repository, but is merely made available for ad-hoc querying.

We are now able to express our query asking for those dataflow executions, whose final result would change if we replaced BLASTv1 by BLASTv2.

```
SELECT O.ID, O.result, E.value
  FROM (SELECT T.value AS result, D.expr, R.vassign, R.btree
          FROM Runs R, Dataflows D, Triples T
         WHERE R.dID=D.dID
           AND R.rID=T.rID AND T.subexpr = 'e1'
           AND XMLEXISTS('$b//ext="BLASTv1"'
                         PASSING R.btree AS "b")) AS O,
       LATERAL (XMLTABLE('copy $newb := $b
                          modify for $n in $newb//extentry
                                 where $n/ext="BLASTv1"
                                 return
                                   replace value of node $n/ext
                                   with "BLASTv2"
                          return $newb'
                          PASSING O.btree AS "b"
                          COLUMNS "newbtree" XML PATH ".") as N,
       TABLE (Eval(O.expr, O.vassign, N.newbtree) AS E
 WHERE E.subexpr='e1' AND is_different(O.result, E.value)
```

The condition `E.subexpr='e1'` on the last line selects the top-level NRC expression so as to retrieve the final result value of each rerun. Note also the use of XQuery Update facilities.

In the above example, we only rewrite the binding trees, not the actual NRC expressions themselves. It should be clear by now that such rewritings are equally possible. For example, we might want to see the effect of shutting out certain parts of certain dataflows. We can express such queries using the same techniques.

4.2 Subvalue provenance in past executions

Recall that a result value of an execution of a dataflow may be quite complex. In such cases it may be desirable to be able to track how a particular occurrence s of a subvalue of that result was produced during an execution. Intuitively, the *subvalue provenance* of s is the restriction of the corresponding past execution to all subexpressions and subvalues of intermediate results, that have, in a sense[†], contributed to the appearance of s in the final result. From this point of view, tracking subvalue provenance can be considered as a special case of querying a past execution.

[†]We argue our understanding of this type of provenance in Section 4.2.3.

4.2.1 Subruns

Before we can define subvalue provenance, we first need the notion of a *subrun*. Intuitively, in Section 3.1.1, we recursively construct a run R of an expression e, from runs of subexpressions of e. Now we need a selection mechanism by which we can extract from R only that part that constitutes the run of a particular subexpression inside R.

Definition 4.1 (Subrun). Let $\sigma, \zeta \approx e \Rightarrow R$ and $(\Phi' \cdot [e'], \sigma') \in SI(R)$. The *subrun of R for* $(\Phi' \cdot [e'], \sigma')$, denoted by $subrun(\Phi' \cdot [e'], \sigma', R)$, is the following set of triples:

$$\{([e'] \cdot \Phi'', \sigma'', v'') \mid (\Phi' \cdot [e'] \cdot \Phi'', \sigma'', v'') \in R \text{ and } \sigma' \text{ is a prefix of } \sigma''\}.$$

Example 4.2. In Example 3.2 (p. 42), we have computed a run R for $e = x \cup \{y.r\}$, with $e_1 = \{y.r\}$, and $e_2 = y.r$. Here, we continue the example and extract subruns of R: $S_1 = subrun([e, 1, x], \sigma, R)$, and $S_2 = subrun([e, 2, e_1], \sigma, R)$.

$$S_1 = \{\, ([x],\, \sigma,\, in_1) \,\}$$

$$\begin{aligned} S_2 = \{\, & ([e_1],\, \sigma,\, v_s), \\ & ([e_1, e_2],\, \sigma,\, v_p), \\ & ([e_1, e_2, y],\, \sigma,\, in_2) \,\} \end{aligned}$$

□

Example 4.3. In Example 3.3 (p. 42), we have computed a run R for

$$e = \text{for } x \text{ in } y \text{ return } \langle b\colon x.b, c\colon f(x.a) \rangle\,,$$

with $e_1 = \langle b\colon e_2, c\colon e_3 \rangle$, $e_2 = x.b$, $e_3 = f(e_4)$, and $e_4 = x.a$. Here, we continue the example and extract subruns of R: $S = subrun([e, 1, y], \sigma, R)$, and $S_{1,1} = subrun([e, 2, e_1], add(\sigma, x, in_1), R)$.

$$S = \{\, ([e, 1, y],\, \sigma,\, in) \,\}$$

$$\begin{aligned} S_{1,1} = \{\, & ([e_1],\, add(\sigma, x, in_1),\, v_1), \\ & ([e_1, b, e_2],\, add(\sigma, x, in_1),\, b_1), \\ & ([e_1, b, e_2, x],\, add(\sigma, x, in_1),\, in_1), \\ & ([e_1, c, e_3],\, add(\sigma, x, in_1),\, w_1), \\ & ([e_1, c, e_3, 1, e_4],\, add(\sigma, x, in_1),\, a_1), \end{aligned}$$

$$([e_1, c, e_3, 1, e_4, x],\ add(\sigma, x, in_1),\ in_1)\ \}$$

We now extract $S'_{1,1} = subrun([e_1, b, e_2], add(\sigma, x, in_1), S_{1,1})$.

$$\begin{aligned} S'_{1,1} = \{\ &([e_2],\ add(\sigma, x, in_1),\ b_1), \\ &([e_2, x],\ add(\sigma, x, in_1),\ in_1)\ \} \end{aligned}$$

□

In the following proposition we prove that subruns are actual runs.

Proposition 4.4. *If* $\sigma, \zeta \approx\models e \Rightarrow R$ *and* $(\Phi' \cdot [e'], \sigma') \in SI(R)$*, then*

$$\sigma', \zeta \approx\models e' \Rightarrow subrun(\Phi' \cdot [e'], \sigma', R)\,.$$

Proof. We prove the proposition by induction on the length of Φ'.[‡]

If the length of Φ' is zero, then Φ' is empty (Definition 2.20) and, by Lemma 3.7, $e' = e$ and $\sigma' = \sigma$. We prove that $subrun([e], \sigma, R) = R$.

Let $([e]{\cdot}\Phi'', \sigma'', v'') \in subrun([e], \sigma, R)$. Then, by Definition 4.1, $([e]{\cdot}\Phi'', \sigma'', v'')$ is a triple in R. Therefore $subrun([e], \sigma, R) \subseteq R$.

Let (Φ'', σ'', v'') be a triple in R. By Lemma 3.8, $\Phi'' = [e] \cdot \Psi$, for some Ψ, and we know that σ is a prefix of σ''. According to Definition 4.1, $([e] \cdot \Psi, \sigma'', v'') \in subrun(e, \sigma, R)$. Therefore $R \subseteq subrun([e], \sigma, R)$.

Let us assume the proposition holds for all Φ' of length n.

We now prove the proposition for Φ' of length $n+1$. By Lemma 3.8, we know that Φ' starts with $[e]$. Let $\Phi' = [e] \cdot \Psi$, for some Ψ of length at most n.

We have to prove that $\sigma', \zeta \approx\models e' \Rightarrow subrun([e] \cdot \Psi \cdot [e'], \sigma', R)$. Therefore we inspect all possible forms of e. (Since Φ' is not empty, e cannot be a constant, a variable, or an empty-set expression.)

1. If e is of the form
   (a) $\{e_1\}$ (Rule R.4),
   (b) $\bigcup e_1$ (Rule R.6),
   (c) $e_1.l$ (Rule R.8), or
   (d) $e_1 = \varnothing$ (Rules R.12–R.13),

[‡]We define the length of a (sub-sequence of a) subexpression path Φ naturally as the number of elements in Φ that are NRC expressions.

then, by Definition 2.20, $\Psi \cdot [e'] \multimapinv e_1$. From the corresponding rules, we know that there is a run R_1 for e_1, such that

$$R = [e] \cdot R_1 \cup \{([e], \sigma, \mathit{result}(R))\}.$$

Since $([e] \cdot \Psi \cdot [e'], \sigma') \in \mathit{SI}(R)$, by Definition 3.6, there is a value v' such that $([e] \cdot \Psi \cdot [e'], \sigma', v') \in R$.

$$\begin{aligned}
&([e] \cdot \Psi \cdot [e'], \sigma', v') \in R \\
&\quad \Longrightarrow ([e] \cdot \Psi \cdot [e'], \sigma', v') \in [e] \cdot R_1 \cup \{([e], \sigma, \mathit{result}(R))\} \\
&\qquad \text{the length of } [e] \cdot \Psi \cdot [e'] \text{ is at least } 2 \\
&\quad \Longrightarrow ([e] \cdot \Psi \cdot [e'], \sigma', v') \in [e] \cdot R_1 \\
&\quad \Longrightarrow (\Psi \cdot [e'], \sigma', v') \in R_1 \\
&\quad \Longrightarrow (\Psi \cdot [e'], \sigma') \in \mathit{SI}(R_1)
\end{aligned}$$

For R_1 and $(\Psi \cdot [e'], \sigma') \in \mathit{SI}(R_1)$, we can apply the induction hypothesis, thus $\sigma', \zeta \models\!\approx e' \Rightarrow \mathit{subrun}(\Psi \cdot [e'], \sigma', R_1)$. It is sufficient to show that $\mathit{subrun}(\Psi \cdot [e'], \sigma', R_1) = \mathit{subrun}([e] \cdot \Psi \cdot [e'], \sigma', R)$.

$$\begin{aligned}
&([e'] \cdot \Psi'', \sigma'', v'') \in \mathit{subrun}(\Psi \cdot [e'], \sigma', R_1) \\
&\qquad \text{applying Definition 4.1, } \sigma' \text{ is a prefix of } \sigma'' \\
&\quad \Longleftrightarrow (\Psi \cdot [e'] \cdot \Psi'', \sigma'', v'') \in R_1 \\
&\quad \Longleftrightarrow ([e] \cdot \Psi \cdot [e'] \cdot \Psi'', \sigma'', v'') \in [e] \cdot R_1 \\
&\qquad \text{the length of } [e] \cdot \Psi \cdot [e'] \text{ is at least } 2 \\
&\quad \Longleftrightarrow ([e] \cdot \Psi \cdot [e'] \cdot \Psi'', \sigma'', v'') \in R \\
&\qquad \text{applying Definition 4.1, } \sigma' \text{ is a prefix of } \sigma'' \\
&\quad \Longleftrightarrow ([e'] \cdot \Psi'', \sigma'', v'') \in \mathit{subrun}([e] \cdot \Psi \cdot [e'], \sigma', R)
\end{aligned}$$

2. If e is of the form
   (a) $e_1 \cup e_2$ (Rule R.5),
   (b) $e_1 = e_2$ (Rules R.10–R.11),
   (c) let $x := e_1$ in e_2 (Rule R.16),
   (d) $e =$ if e_0 then e_1 else e_2 (Rules R.14–R.15),
   (e) $\langle l_1 : e_1, \ldots, l_n : e_n \rangle$ (Rule R.7), or
   (f) $f(e_1, \ldots, e_n)$ (Rule R.17),

   then, by Definition 2.20, $\Psi = [m_i] \cdot \Psi_i$ where $\Psi_i \cdot [e'] \multimapinv e_i$, for (a–c) $i \in \{1,2\}$, with $m_i = i$;

(d) $i \in \{0,1,2\}$, with $m_i = i$;

(e) $i \in \{1,\dots,n\}$, with $m_i = l_i$;

(f) $i \in \{1,\dots,n\}$, with $m_i = i$.

Let

(a–c) $I = \{1,2\}$;

(d) $I = \{0,1\}$ if $\mathit{result}(R_0) = \mathtt{true}$; $I = \{0,2\}$ if $\mathit{result}(R_0) = \mathtt{false}$;

(e–f) $I = \{1,\dots,n\}$.

Then, by the corresponding rules, we know that there are runs R_i for e_i, for $i \in I$, such that

$$R = \bigcup_{i\in I} [e, m_i] \cdot R_i \,\cup\, \{([e], \sigma, \mathit{result}(R))\}\,.$$

Let $j \in I$. Since $([e, m_j] \cdot \Psi_j \cdot [e'], \sigma') \in \mathit{SI}(R)$, by Definition 3.6, there is a value v' such that $([e, m_j] \cdot \Psi_j \cdot [e'], \sigma', v') \in R$.

$$\begin{aligned}
&([e, m_j] \cdot \Psi_j \cdot [e'], \sigma', v') \in R\\
&\qquad \text{the prefix } [e, m_j] \text{ must match}\\
&\quad\implies ([e, m_j] \cdot \Psi_j \cdot [e'], \sigma', v') \in [e, m_j] \cdot R_j\\
&\quad\implies (\Psi_j \cdot [e'], \sigma', v') \in R_j\\
&\quad\implies (\Psi_j \cdot [e'], \sigma') \in \mathit{SI}(R_j)
\end{aligned}$$

For R_j and $(\Psi_j \cdot [e'], \sigma') \in \mathit{SI}(R_j)$, we can apply the induction hypothesis, thus $\sigma', \zeta \models_{\approx} e' \Rightarrow \mathit{subrun}(\Psi_j \cdot [e'], \sigma', R_j)$. It is sufficient to show that $\mathit{subrun}(\Psi_j \cdot [e'], \sigma', R_j) = \mathit{subrun}([e] \cdot \Psi \cdot [e'], \sigma', R)$.

$$\begin{aligned}
&([e'] \cdot \Psi'', \sigma'', v'') \in \mathit{subrun}(\Psi_j \cdot [e'], \sigma', R_j)\\
&\qquad \text{applying Definition 4.1, } \sigma' \text{ is a prefix of } \sigma''\\
&\quad\iff (\Psi_j \cdot [e'] \cdot \Psi'', \sigma'', v'') \in R_j\\
&\quad\iff ([e, m_j] \cdot \Psi_j \cdot [e'] \cdot \Psi'', \sigma'', v'') \in [e, m_j] \cdot R_j\\
&\qquad \text{the prefix } [e, m_j] \text{ must match}\\
&\quad\iff ([e, m_j] \cdot \Psi_j \cdot [e'] \cdot \Psi'', \sigma'', v'') \in R\\
&\qquad \text{applying Definition 4.1, } \sigma' \text{ is a prefix of } \sigma''\\
&\quad\iff ([e'] \cdot \Psi'', \sigma'', v'') \in \mathit{subrun}([e, m_j] \cdot \Psi_j \cdot [e'], \sigma', R)\\
&\quad\iff ([e'] \cdot \Psi'', \sigma'', v'') \in \mathit{subrun}([e] \cdot \Psi \cdot [e'], \sigma', R)
\end{aligned}$$

3. If $e = \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2$, then, by Definition 2.20, $\Psi = [i] \cdot \Psi_i$ where $\Psi_i \cdot [e'] \multimap e_i$, with $i \in \{1, 2\}$.

   From Rule R.9, we know that there is a run R_1 for e_1, and for $w \in \mathit{result}(R_1)$, there are runs R_w for e_2, such that

$$R = [e,1] \cdot R_1 \cup (\textstyle\bigcup_w [e,2] \cdot R_w) \cup \{([e], \sigma, \mathit{result}(R))\}.$$

   Case $\Psi = [1] \cdot \Psi_1$ Since $([e,1] \cdot \Psi_1 \cdot [e'], \sigma') \in \mathit{SI}(R)$, by Definition 3.6, there is a value v' such that $([e,1] \cdot \Psi_1 \cdot [e'], \sigma', v') \in R$.

$$\begin{aligned}
&([e,1] \cdot \Psi_1 \cdot [e'], \sigma', v') \in R \\
&\qquad \text{the prefix } [e,1] \text{ must match} \\
&\quad \Longrightarrow ([e,1] \cdot \Psi_1 \cdot [e'], \sigma', v') \in [e,1] \cdot R_1 \\
&\quad \Longrightarrow (\Psi_1 \cdot [e'], \sigma', v') \in R_1 \\
&\quad \Longrightarrow (\Psi_1 \cdot [e'], \sigma') \in \mathit{SI}(R_1)
\end{aligned}$$

   For R_1 and $(\Psi_1 \cdot [e'], \sigma') \in \mathit{SI}(R_1)$, we can apply the induction hypothesis, thus $\sigma', \zeta \approx\!\models e' \Rightarrow \mathit{subrun}(\Psi_1 \cdot [e'], \sigma', R_1)$. It is sufficient to show that $\mathit{subrun}(\Psi_1 \cdot [e'], \sigma', R_1) = \mathit{subrun}([e] \cdot \Psi \cdot [e'], \sigma', R)$.

$$\begin{aligned}
&([e'] \cdot \Psi'', \sigma'', v'') \in \mathit{subrun}(\Psi_1 \cdot [e'], \sigma', R_1) \\
&\qquad \text{applying Definition 4.1, } \sigma' \text{ is a prefix of } \sigma'' \\
&\quad \Longleftrightarrow (\Psi_1 \cdot [e'] \cdot \Psi'', \sigma'', v'') \in R_1 \\
&\quad \Longleftrightarrow ([e,1] \cdot \Psi_1 \cdot [e'] \cdot \Psi'', \sigma'', v'') \in [e,1] \cdot R_1 \\
&\qquad \text{the prefix } [e,1] \text{ must match} \\
&\quad \Longleftrightarrow ([e,1] \cdot \Psi_1 \cdot [e'] \cdot \Psi'', \sigma'', v'') \in R \\
&\qquad \text{applying Definition 4.1, } \sigma' \text{ is a prefix of } \sigma'' \\
&\quad \Longleftrightarrow ([e'] \cdot \Psi'', \sigma'', v'') \in \mathit{subrun}([e,1] \cdot \Psi_1 \cdot [e'], \sigma', R) \\
&\quad \Longleftrightarrow ([e'] \cdot \Psi'', \sigma'', v'') \in \mathit{subrun}([e] \cdot \Psi \cdot [e'], \sigma', R)
\end{aligned}$$

   Case $\Psi = [2] \cdot \Psi_2$ Since $([e,2] \cdot \Psi_2 \cdot [e'], \sigma') \in \mathit{SI}(R)$, by Definition 3.6, there is a value v' such that $([e,2] \cdot \Psi_2 \cdot [e'], \sigma', v') \in R$.

$$\begin{aligned}
&([e,2] \cdot \Psi_2 \cdot [e'], \sigma', v') \in R \\
&\qquad \text{the prefix } [e,2] \text{ must match}
\end{aligned}$$

$$\begin{aligned}
&\Longrightarrow ([e,2]\cdot\Psi_2\cdot[e'],\sigma',v') \in \textstyle\bigcup_w [e,2]\cdot R_w \\
&\quad \text{Lemma 3.9, } \exists!\, u \in \mathit{result}(R_1) \\
&\Longrightarrow ([e,2]\cdot\Psi_2\cdot[e'],\sigma',v') \in [e,2]\cdot R_u \\
&\Longrightarrow (\Psi_2\cdot[e'],\sigma',v') \in R_u \\
&\Longrightarrow (\Psi_2\cdot[e'],\sigma') \in \mathit{SI}(R_u)
\end{aligned}$$

For R_u and $(\Psi_2\cdot[e'],\sigma') \in \mathit{SI}(R_u)$, we can apply the induction hypothesis, thus $\sigma',\zeta \approx\!\!\models e' \Rightarrow \mathit{subrun}(\Psi_2\cdot[e'],\sigma',R_u)$. It is sufficient to show that $\mathit{subrun}(\Psi_2\cdot[e'],\sigma',R_u) = \mathit{subrun}([e]\cdot\Psi\cdot[e'],\sigma',R)$.

$$\begin{aligned}
&([e']\cdot\Psi'',\sigma'',v'') \in \mathit{subrun}(\Psi_2\cdot[e'],\sigma',R_u) \\
&\quad \text{applying Definition 4.1, } \sigma' \text{ is a prefix of } \sigma'' \\
&\Longleftrightarrow (\Psi_2\cdot[e']\cdot\Psi'',\sigma'',v'') \in R_u \\
&\Longleftrightarrow ([e,2]\cdot\Psi_2\cdot[e']\cdot\Psi'',\sigma'',v'') \in [e,2]\cdot R_u \\
&\quad \text{Lemma 3.9, } \exists!\, u \in \mathit{result}(R_1) \\
&\Longleftrightarrow ([e,2]\cdot\Psi_2\cdot[e']\cdot\Psi'',\sigma'',v'') \in R \\
&\quad \text{applying Definition 4.1, } \sigma' \text{ is a prefix of } \sigma'' \\
&\Longleftrightarrow ([e']\cdot\Psi'',\sigma'',v'') \in \mathit{subrun}([e,2]\cdot\Psi_2\cdot[e'],\sigma',R) \\
&\Longleftrightarrow ([e']\cdot\Psi'',\sigma'',v'') \in \mathit{subrun}([e]\cdot\Psi\cdot[e'],\sigma',R)
\end{aligned}$$

□

4.2.2 The subvalue provenance function

Let e be an NRC expression, σ a value assignment over $FV(e)$, and ζ a function assignment over $SN(e)$. Let $\sigma,\zeta \approx\!\!\models e \Rightarrow R$ and $v = \mathit{result}(R)$. Let φ be an occurrence of a subvalue of v, i.e., $\varphi \leftarrow\!\!\bullet\, v$.§

Recall that we represent R as a set of triples of the form (Φ',σ',v'), with Φ' an occurrence of subexpression e' of e, such that $\sigma',\zeta \models e' \Rightarrow v'$. Likewise, we represent the *subvalue provenance P of φ in R* as a set consisting of triples. Each triple is of the form (Φ',σ',φ'), where, again, Φ' is an occurrence of subexpression e' of e, with $\sigma',\zeta \models e' \Rightarrow v'$, and where φ' is an occurrence of a subvalue of v', i.e., $\varphi' \leftarrow\!\!\bullet\, v'$. Such a triple represents the information that the intermediate result v' has partly contributed to φ in the output; φ' then indicates which part. In particular, P always contains the triple $([e],\sigma,\varphi)$.

§Remember, for complex values, we identify an occurrence of a subvalue by a subvalue path ending in that particular subvalue (Definition 2.3).

In the following Rules P.1–P.18, we define the *subvalue provenance function*, denoted by *Prov*(). In general, we refer to sets produced by *Prov*() as *provenance sets.* As some rules[¶] are quite complex, we provide an informal explanation when necessary. We use symbol ":=" in the premises to indicate that the notation on the left-hand side is an abbreviation for the right-hand side. In each rule, $Prov(\varphi, R)$ is *the subvalue provenance of* φ *in* R. In the following, we frequently abbreviate "subvalue path" to simply "subvalue".

Rules P.1–P.3 deal with the base NRC expressions: a constant, a variable and an empty-set expression. A run of these expressions is always a singleton. In a provenance set for a constant or an empty-set expression, we use the triple from the run, and, in the third component, replace the value by its trivial subvalue. Likewise, for a variable, we replace the value v by its subvalue φ. In this case φ may also be a proper subvalue of v.

$$\frac{\sigma, \zeta \approx\!\!\models \mathtt{a} \Rightarrow R \qquad R = \{([\mathtt{a}], \sigma, \mathtt{a})\}}{Prov([\mathtt{a}], R) \overset{def}{=} \{([\mathtt{a}],\ \sigma,\ [\mathtt{a}])\}}\ \text{P.1}$$

$$\frac{\sigma, \zeta \approx\!\!\models x \Rightarrow R \qquad R = \{([x], \sigma, v)\} \qquad \varphi \leftarrow\!\!\bullet\ v}{Prov(\varphi, R) \overset{def}{=} \{([x],\ \sigma,\ \varphi)\}}\ \text{P.2}$$

$$\frac{\sigma, \zeta \approx\!\!\models \varnothing \Rightarrow R \qquad R = \{([\varnothing], \sigma, \emptyset)\}}{Prov([\emptyset], R) \overset{def}{=} \{([\varnothing],\ \sigma,\ [\emptyset])\}}\ \text{P.3}$$

For the following NRC expressions: a set-expression, a union, a flatten, and a tuple-expression; we have separate rules for the trivial subvalue (P.4,P.7,P.9), and for proper subvalues of the final result (P.5,P.6,P.8,P.10). Both kinds of rules call *Prov*() recursively on certain subruns of R, namely, the subruns of the direct subexpressions that have contributed to φ. The rules for the trivial subvalue simply call *Prov*(), for each subrun, for the trivial subvalue of the final result of this subrun. The rules for proper subvalues, however, call *Prov*(), for each subrun, for proper subvalues of the final result of the subrun, but only for these subvalues that have actually contributed to φ. Note that the construction of the proper subvalue of a subrun depends on the corresponding

[¶]Again, we intensively use the prefixing operation defined in Eq. A.1. The rules should be read with the operator "·" having a higher precedence than the operator "∪".

subexpression.

$$\dfrac{\begin{array}{c} e = \{e'\} \text{ or } e = \bigcup e' \\ \sigma, \zeta \mathrel{|\!\!\approx} e \Rightarrow R \qquad v := \mathit{result}(R) \qquad S := \mathit{subrun}([e, e'], \sigma, R) \end{array}}{\mathit{Prov}([v], R) \overset{\mathit{def}}{=} [e] \cdot \mathit{Prov}([\mathit{result}(S)], S) \,\cup\, \{([e],\, \sigma,\, [v])\}} \;\text{P.4}$$

$$\dfrac{e = \{e'\} \qquad \sigma, \zeta \mathrel{|\!\!\approx} e \Rightarrow R \qquad \varphi \mathrel{\leftarrow\!\!\dot{\bullet}} \mathit{result}(R) \qquad \varphi = [\mathit{result}(R)] \cdot \varphi'}{\mathit{Prov}(\varphi, R) \overset{\mathit{def}}{=} [e] \cdot \mathit{Prov}(\varphi', \mathit{subrun}([e, e'], \sigma, R)) \,\cup\, \{([e],\, \sigma,\, \varphi)\}} \;\text{P.5}$$

$$\dfrac{\begin{array}{c} e = \bigcup e' \qquad \sigma, \zeta \mathrel{|\!\!\approx} e \Rightarrow R \qquad \varphi \mathrel{\leftarrow\!\!\dot{\bullet}} \mathit{result}(R) \qquad \varphi = [\mathit{result}(R)] \cdot \varphi' \\ \varphi' \mathrel{\leftarrow\!\!\bullet} u \qquad S := \mathit{subrun}([e, e'], \sigma, R) \qquad W := \{w \in \mathit{result}(S) \mid u \in w\} \end{array}}{\mathit{Prov}(\varphi, R) \overset{\mathit{def}}{=} \bigcup_{w \in W} [e] \cdot \mathit{Prov}([\mathit{result}(S), w] \cdot \varphi', S) \,\cup\, \{([e],\, \sigma,\, \varphi)\}} \;\text{P.6}$$

$$\dfrac{\begin{array}{c} e = (e_1 \cup e_2) \qquad \sigma, \zeta \mathrel{|\!\!\approx} e \Rightarrow R \\ v := \mathit{result}(R) \qquad \forall i \in \{1, 2\} \colon S_i := \mathit{subrun}([e, i, e_i], \sigma, R) \end{array}}{\mathit{Prov}([v], R) \overset{\mathit{def}}{=} \bigcup_{i \in \{1,2\}} [e, i] \cdot \mathit{Prov}([\mathit{result}(S_i)], S_i) \,\cup\, \{([e],\, \sigma,\, [v])\}} \;\text{P.7}$$

$$\dfrac{\begin{array}{c} e = (e_1 \cup e_2) \\ \sigma, \zeta \mathrel{|\!\!\approx} e \Rightarrow R \qquad \varphi \mathrel{\leftarrow\!\!\dot{\bullet}} \mathit{result}(R) \qquad \varphi = [\mathit{result}(R)] \cdot \varphi' \\ \forall i \in \{1, 2\} \colon S_i := \mathit{subrun}([e, i, e_i], \sigma, R) \wedge v_i := \mathit{result}(S_i) \\ I := \{i \in \{1, 2\} \mid [v_i] \cdot \varphi' \mathrel{\leftarrow\!\!\bullet} v_i\} \end{array}}{\mathit{Prov}(\varphi, R) \overset{\mathit{def}}{=} \bigcup_{i \in I} [e, i] \cdot \mathit{Prov}([v_i] \cdot \varphi', S_i) \,\cup\, \{([e],\, \sigma,\, \varphi)\}} \;\text{P.8}$$

$$\dfrac{\begin{array}{c} e = \langle l_1 \colon e_1, \ldots, l_n \colon e_n \rangle \qquad \sigma, \zeta \mathrel{|\!\!\approx} e \Rightarrow R \\ v := \mathit{result}(R) \qquad \forall i \in \{1, \ldots, n\} \colon S_i := \mathit{subrun}([e, l_i, e_i], \sigma, R) \end{array}}{\mathit{Prov}([v], R) \overset{\mathit{def}}{=} \bigcup_{i \in \{1, \ldots, n\}} [e, l_i] \cdot \mathit{Prov}([\mathit{result}(S_i)], S_i) \,\cup\, \{([e],\, \sigma,\, [v])\}} \;\text{P.9}$$

$$\frac{e = \langle l_1 : e_1, \ldots, l_n : e_n \rangle \qquad \sigma, \zeta \approx\!\models e \Rightarrow R \qquad \varphi \leftarrow\!\!\bullet\ \mathit{result}(R) \\ \varphi = [\mathit{result}(R), l_i] \cdot \varphi' \qquad S := \mathit{subrun}([e, l_i, e_i], \sigma, R)}{\mathit{Prov}(\varphi, R) \overset{\mathit{def}}{=} [e, l_i] \cdot \mathit{Prov}(\varphi', S) \cup \{([e],\ \sigma,\ \varphi)\}} \ \text{P.10}$$

In Rules P.11–P.15 for the following NRC expressions: a projection, an equality test, an emptiness test, a service-call expression, and an if-expression, we do not need to distinguish between trivial and proper subvalues. In Rules P.12–P.13 we do not even recursively call *Prov*() at all. These expressions do not assemble their final results from the final results of the subruns of their direct subexpressions. Indeed, for an equality or an emptiness test, the final result has been created by the expression, and has only the trivial subvalue. We cannot trace that subvalue any further. For a service-call expression, the final result value has been provided by the service. Here, both trivial and proper subvalues of the final result are possible. We cannot trace these subvalues any further because, in general, the service call is considered a black box. (In some cases, however, there may be sufficient information to whiten a black box a little, see Section 4.2.4.)

$$\frac{e = e'.l \\ \sigma, \zeta \approx\!\models e \Rightarrow R \qquad \varphi \leftarrow\!\!\bullet\ \mathit{result}(R) \qquad S := \mathit{subrun}([e, e'], \sigma, R)}{\mathit{Prov}(\varphi, R) \overset{\mathit{def}}{=} [e] \cdot \mathit{Prov}([\mathit{result}(S), l] \cdot \varphi, S) \cup \{([e],\ \sigma,\ \varphi)\}} \ \text{P.11}$$

$$\frac{e = (e_1 = e_2) \text{ or } e = (e' = \varnothing) \qquad \sigma, \zeta \approx\!\models e \Rightarrow R \qquad v := \mathit{result}(R)}{\mathit{Prov}([v], R) \overset{\mathit{def}}{=} \{([e],\ \sigma,\ [v])\}} \ \text{P.12}$$

$$\frac{e = f(e_1, \ldots, e_n) \qquad \sigma, \zeta \approx\!\models e \Rightarrow R \qquad \varphi \leftarrow\!\!\bullet\ \mathit{result}(R)}{\mathit{Prov}(\varphi, R) \overset{\mathit{def}}{=} \{([e],\ \sigma,\ \varphi)\}} \ \text{P.13}$$

In Rules P.14–P.15, function *Prov*() is recursively called for the same subvalue and only for the subrun that corresponds to the actually executed branch of the expression.

$$\frac{e = \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \qquad \sigma, \zeta \approx\!\models e \Rightarrow R \\ \varphi \leftarrow\!\!\bullet\ \mathit{result}(R) \qquad \mathit{result}(\mathit{subrun}([e, 0, e_0], \sigma, R)) = \mathtt{true}}{\mathit{Prov}(\varphi, R) \overset{\mathit{def}}{=} [e, 1] \cdot \mathit{Prov}(\varphi, \mathit{subrun}([e, 1, e_1], \sigma, R)) \cup \{([e],\ \sigma,\ \varphi)\}} \ \text{P.14}$$

$$\frac{e = \textsf{if } e_0 \textsf{ then } e_1 \textsf{ else } e_2 \qquad \sigma, \zeta \models\approx e \Rightarrow R \qquad \varphi \leftarrow\!\!\bullet\ \mathit{result}(R) \qquad \mathit{result}(\mathit{subrun}([e,0,e_0],\sigma,R)) = \texttt{false}}{\mathit{Prov}(\varphi,R) \stackrel{\mathit{def}}{=} [e,2]\cdot \mathit{Prov}(\varphi, \mathit{subrun}([e,2,e_2],\sigma,R)) \cup \{([e],\ \sigma,\ \varphi)\}}\ \text{P.15}$$

The last three rules are the most complex ones. They are for the following cases: Rule P.16 for a let-expression; Rule P.17 for a for-expression and the trivial subvalue of the final result; and Rule P.18 for a for-expression and a proper subvalue of the final result. Here, we also need to consider the bound variables of the expressions.

We first introduce an auxiliary function, which selects provenance-triples with a subexpression path ending in a particular variable.

Definition 4.5. Let X, Y, and Z be sets. For a set $P \subseteq \mathit{SEQ}(X) \times Y \times Z$, for $x \in X$ and $y \in Y$, we define:

$$\mathit{Sel}(x,y,P) \stackrel{\mathit{def}}{=} \{(q,y,z) \in P \mid x \text{ is the } \textit{last} \text{ element of } q\}\ .$$

For a let-expression, we first recursively call $\mathit{Prov}()$ on the same subvalue for the body of the expression. In the resulting provenance set, we look for triples where the subexpression path in the first component ends in the bound variable.[‖] Also, the second component of the triple must contain the same value assignment. Then, for each of these triples, we call $\mathit{Prov}()$ on the subvalue from the third component, for the head of the expression.

$$\frac{\begin{array}{c} e = \textsf{let } x := e_1 \textsf{ in } e_2 \qquad \sigma,\zeta \models\approx e \Rightarrow R \qquad \varphi \leftarrow\!\!\bullet\ \mathit{result}(R) \\ S := \mathit{subrun}([e,1,e_1],\sigma,R) \qquad \sigma' := \mathit{add}(\sigma,x,\mathit{result}(S)) \\ P := \mathit{Prov}(\varphi, \mathit{subrun}([e,2,e_2],\sigma',R)) \qquad \mathbb{O} := \mathit{Sel}(x,\sigma',P) \end{array}}{\mathit{Prov}(\varphi,R) \stackrel{\mathit{def}}{=} [e,2]\cdot P \cup \{([e],\ \sigma,\ \varphi)\} \cup \bigcup_{(\Phi,\sigma',\varphi')\in\mathbb{O}} [e,1]\cdot \mathit{Prov}(\varphi',S)}\ \text{P.16}$$

For a for-expression and the trivial subvalue of a final result, we first recursively call $\mathit{Prov}()$ for each subrun of the body, on the trivial subvalue of the final result of that subrun. Then, for each resulting provenance set, we process the

[‖]We have formerly defined the subvalue provenance without tracking the bound variables [HKS$^+$07]. We owe this improvement to our rules to feedback provided by James Cheney, University of Edinburgh.

bound variable in a similar way as for a let-expression.

$$\frac{\begin{gathered} e = \textsf{for } x \textsf{ in } e_1 \textsf{ return } e_2 \\ \sigma, \zeta \models\!\!\approx e \Rightarrow R \qquad S := \mathit{subrun}([e, 1, e_1], \sigma, R) \\ \forall w \in \mathit{result}(S)\colon \sigma_w := \mathit{add}(\sigma, x, w) \wedge S_w := \mathit{subrun}([e, 2, e_2], \sigma_w, R) \\ \wedge\ P_w := \mathit{Prov}([\mathit{result}(S_w)], S_w) \wedge \mathbb{O}_w := \mathit{Sel}(x, \sigma_w, P_w) \end{gathered}}{\begin{aligned} \mathit{Prov}([\mathit{result}(R)], R) \overset{\mathit{def}}{=} & \bigcup_{w \in \mathit{result}(S)} [e, 2] \cdot P_w \cup \{([e],\ \sigma,\ [\mathit{result}(R)])\} \cup \\ & \bigcup_{w \in \mathit{result}(S)} \ \bigcup_{(\Phi, \sigma_w, \varphi) \in \mathbb{O}_w} [e, 1] \cdot \mathit{Prov}([\mathit{result}(S)] \cdot \varphi, S) \end{aligned}} \ \text{P.17}$$

For a for-expression and a proper subvalue of a final result, we recursively call *Prov*() only for those subruns of the body that have contributed to the subvalue. Again, for each resulting provenance set, we process the bound variable in a similar way as for a let-expression.

$$\frac{\begin{gathered} e = \textsf{for } x \textsf{ in } e_1 \textsf{ return } e_2 \qquad \sigma, \zeta \models\!\!\approx e \Rightarrow R \qquad \varphi \leftarrow\!\dot{\bullet}\ \mathit{result}(R) \\ \varphi = [\mathit{result}(R)] \cdot \varphi' \qquad \varphi' \leftarrow\!\bullet\ u \qquad S := \mathit{subrun}([e, 1, e_1], \sigma, R) \\ \forall w \in \mathit{result}(S)\colon \sigma_w := \mathit{add}(\sigma, x, w) \wedge S_w := \mathit{subrun}([e, 2, e_2], \sigma_w, R) \\ W := \{w \in \mathit{result}(S) \mid \mathit{result}(S_w) = u\} \\ \forall w \in W\colon P_w = \mathit{Prov}(\varphi', S_w) \wedge \mathbb{O}_w := \mathit{Sel}(x, \sigma_w, P_w) \end{gathered}}{\begin{aligned} \mathit{Prov}(\varphi, R) \overset{\mathit{def}}{=} & \bigcup_{w \in W} [e, 2] \cdot P_w \cup \{([e],\ \sigma,\ \varphi)\} \cup \\ & \bigcup_{w \in W} \ \bigcup_{(\Phi, \sigma_w, \varphi'') \in \mathbb{O}_w} [e, 1] \cdot \mathit{Prov}([\mathit{result}(S)] \cdot \varphi'', S) \end{aligned}} \ \text{P.18}$$

Observe that we can use the rules to compute the subvalue provenance of any subvalue of any intermediate result in R. Indeed, for a triple $(\Phi', \sigma', v') \in R$ and $\varphi' \leftarrow\!\bullet\ v'$, $\mathit{Prov}(\varphi', \mathit{subrun}(\Phi', \sigma', R))$ computes the subvalue provenance in the relevant subrun of R, and the missing prefixes from the subexpression invocation (Φ', σ') can be added afterwards.

Example 4.6. In Example 3.2 (p. 42), we have computed a run R for $e = x \cup \{y.r\}$, with $e_1 = \{y.r\}$, and $e_2 = y.r$. In Example 4.2 (p. 75), we have extracted subruns S_1 and S_2. Here, we continue the example, and track the subvalue provenance of $\varphi = [\mathit{out}, \langle a\colon 5, b\colon 25\rangle, b, 25]$ in R.

We first apply Rule P.8:

$$\mathit{Prov}(\varphi, R) = [e, 1] \cdot \mathit{Prov}([\mathit{in}_1] \cdot \varphi', S_1) \cup [e, 2] \cdot \mathit{Prov}([v_s] \cdot \varphi', S_2) \cup \{([e], \sigma, \varphi)\},$$

with $\varphi' = [\langle a\colon 5, b\colon 25\rangle, b, 25]$. In the first two rows of Figure 4.1, we see how

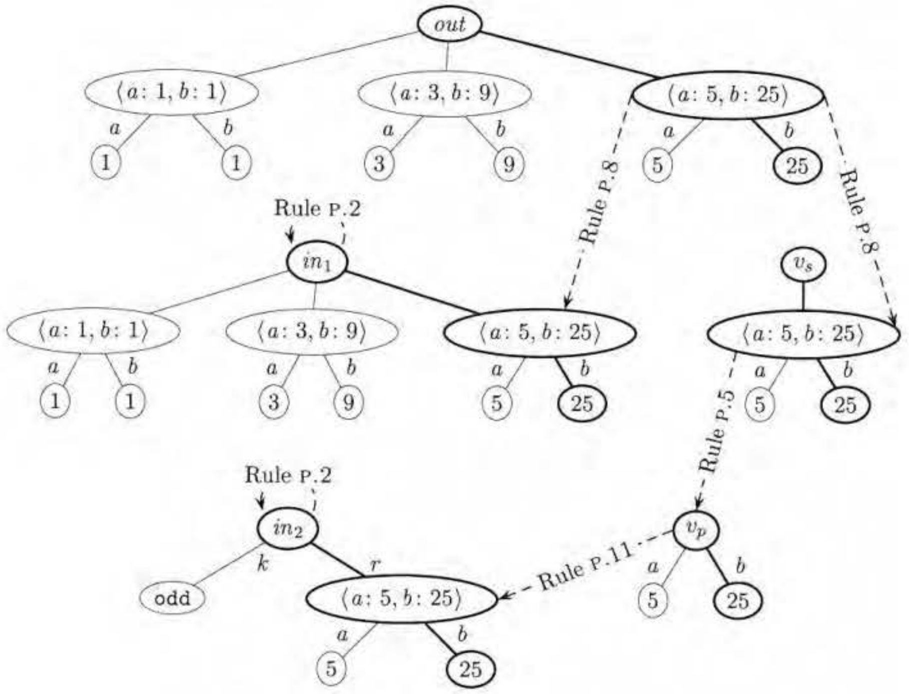


Figure 4.1: Application of subvalue provenance rules in Example 4.6

Rule P.8 maps φ' between the final result *out* of run R, and the final result values in_1 and v_s of, respectively, runs S_1 and S_2.** The subvalue path in the first argument of *Prov*() is marked by bold lines: (i) in value *out* we see φ from $Prov(\varphi, R)$; (ii) in value in_1 we see $[in_1]\cdot\varphi'$ from $Prov([in_1]\cdot\varphi', S_1)$; and (iii) in value v_s we see $[v_s]\cdot\varphi'$ from $Prov([v_s]\cdot\varphi', S_2)$.

We recursively apply, respectively, Rules P.2 and P.5:

$$Prov([in_1]\cdot\varphi', S_1) = \{([x], \sigma, [in_1]\cdot\varphi')\}$$

$$Prov([v_s]\cdot\varphi', S_2) = [e_1]\cdot Prov(\varphi', S_3) \cup \{([e_1], \sigma, [v_s]\cdot\varphi')\},$$

with $S_3 = subrun([e_1, e_2], \sigma, S_2)$. We follow with Rule P.11:

$$Prov(\varphi', S_3) = [e_2]\cdot Prov([in_2, r]\cdot\varphi', S_4) \cup \{([e_2], \sigma, [v_p, b, 25])\},$$

with $S_4 = subrun([e_2, y], \sigma, S_3)$. In the last two rows of Figure 4.1, we see how Rule P.5 maps φ' between v_s and v_p (the final result of run S_3); and Rule P.11 between v_p and in_2 (the final result of run S_4). The subvalue paths

**We use the tree-representation of a complex value from Section 2.1.2.

shown in v_p and in_2 are first arguments in, respectively, $Prov(\varphi', S_3)$ and $Prov([in_2, r] \cdot \varphi', S_4)$.

Finally, we again apply Rule P.2:

$$Prov([in_2, r] \cdot \varphi', S_4) = \{([y], \sigma, [in_2, r] \cdot \varphi')\}$$

After all the necessary concatenations and unions, we obtain the following provenance set:

$$\begin{aligned} Prov(\varphi, R) = \{ & ([e],\ \sigma,\ [out, \langle a\colon 5, b\colon 25\rangle, b, 25]), \\ & ([e, 1, x],\ \sigma,\ [in_1, \langle a\colon 5, b\colon 25\rangle, b, 25]), \\ & ([e, 2, e_1],\ \sigma,\ [v_s, \langle a\colon 5, b\colon 25\rangle, b, 25]), \\ & ([e, 2, e_1, e_2],\ \sigma,\ [v_p, b, 25]) \\ & ([e, 2, e_1, e_2, y],\ \sigma,\ [in_2, r, \langle a\colon 5, b\colon 25\rangle, b, 25]) \} \end{aligned}$$

□

Example 4.7. In Example 3.3 (p. 42), we have computed a run R for

$$e = \mathsf{for}\ x\ \mathsf{in}\ y\ \mathsf{return}\ \langle b\colon x.b, c\colon f(x.a)\rangle\,,$$

with $e_1 = \langle b\colon e_2, c\colon e_3\rangle$, $e_2 = x.b$, $e_3 = f(e_4)$, and $e_4 = x.a$. In Example 4.3 (p. 75), we have extracted subruns S, $S_{1,1}$, and $S'_{1,1}$. Here, we continue the example. Assume that service-call expression e_3 represents an external service.

Let $\varphi_1 = [out, \langle b\colon 4, c\colon 1\rangle, b, 4]$ and $\varphi_2 = [out, \langle b\colon 4, c\colon 1\rangle, c, 1]$. We want to compute both $Prov(\varphi_1, R)$ and $Prov(\varphi_2, R)$.

As a first step, we need to apply Rule (P.18). We see in the premises that we have yet to determine the set W. For $Prov(\varphi_1, R)$, φ'_1 is $[\langle b\colon 4, c\colon 1\rangle, b, 4]$. It is clear that φ'_1 is a subvalue path of both v_1 and v_3. For $Prov(\varphi_2, R)$, φ'_2 is $[\langle b\colon 4, c\colon 1\rangle, c, 1]$. It is also clear that φ'_2 is a subvalue path of both v_1 and v_3. In both cases $W = \{in_1, in_3\}$.

For $i \in \{1, 3\}$ and $S_{1,3} = subrun([e, 2, e_1], add(\sigma, x, in_3), R)$, we now compute $Prov(\varphi'_1, S_{1,i})$. From Rule P.10:

$$Prov(\varphi'_1, S_{1,i}) = [e_1, b] \cdot Prov([4], S'_{1,i}) \cup ([e_1], add(\sigma, x, in_i), [v_i, b, 4])$$

with $S'_{1,3} = subrun([e_1, b, e_2], add(\sigma, x, in_3), S_{1,3})$. We apply Rule P.11:

$$Prov([4], S'_{1,i}) = [e_2] \cdot Prov([in_i, b, 4], S''_{1,i}) \cup ([e_2], add(\sigma, x, in_i), [b_i])$$

with $S''_{1,i} = subrun([e_2, x], add(\sigma, x, in_i), S'_{1,i})$. In Figure 4.2, we see how Rule

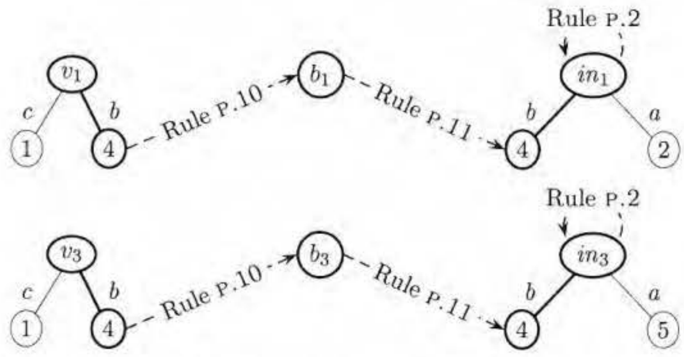


Figure 4.2: Application of subvalue provenance rules in Example 4.7, in computation of $Prov(\varphi'_1, S_{1,1})$ and $Prov(\varphi'_1, S_{1,3})$.

P.10 maps [4] between the final result v_i of run $S_{1,i}$ and the final result b_i of run $S'_{1,i}$. Then Rule P.11 maps [4] between b_i and in_i, the final result of run $S''_{1,i}$. By Rule P.2,

$$Prov([in_i, b, 4], S''_{1,i}) = \{([x], add(\sigma, x, in_i), [in_i, b, 4])\}.$$

All together we obtain,

$$\begin{aligned} Prov(\varphi'_1, S_{1,i}) = \{\, & ([e_1],\ add(\sigma, x, in_i),\ [v_i, b, 4]), \\ & ([e_1, b, e_2],\ add(\sigma, x, in_i),\ [b_i]), \\ & ([e_1, b, e_2, x],\ add(\sigma, x, in_i),\ [in_i, b, 4])\,\} \end{aligned}$$

For $i \in \{1, 3\}$, we compute $Prov(\varphi'_2, S_{2,i})$, with $S_{2,i} = S_{1,i}$. From Rule P.10:

$$Prov(\varphi'_2, S_{2,i}) = [e_1, c] \cdot Prov([1], S'_{2,i}) \cup ([e_1], add(\sigma, x, in_i), \varphi'_2)$$

with $S'_{2,i} = subrun([e_1, c, e_3], add(\sigma, x, in_i), S_{2,i})$. We apply Rule P.13:

$$Prov([1], S'_{2,i}) = \{([e_3], add(\sigma, x, in_i), [w_i])\}.$$

All together we obtain,

$$\begin{aligned} Prov(\varphi'_2, S_{2,i}) = \{\, & ([e_1],\ add(\sigma, x, in_i),\ [v_i, c, 1]) \\ & ([e_1, c, e_3],\ add(\sigma, x, in_i),\ [w_i])\,\} \end{aligned}$$

We can now determine the last sets in the premises of Rule (P.18). For $Prov(\varphi'_1, S_{1,i})$, we have $\mathbb{O}_{1,i} = \{([e_1, b, e_2, x], add(\sigma, x, in_i), [in_i, b, 4])\}$, but for $Prov(\varphi'_2, S_{2,i})$, the sets $\mathbb{O}_{2,i}$ are empty.

We can already compute $Prov(\varphi_2, R)$:

$$Prov(\varphi_2, R) = \bigcup_{i \in \{1,3\}} [e, 2] \cdot Prov(\varphi_2', S_{2,i}) \cup \{([e], \sigma, \varphi_2)\},$$

therefore

$$\begin{aligned} Prov(\varphi_2, R) = \{ & ([e],\ \sigma,\ [out, \langle b\colon 4, c\colon 1\rangle, c, 1]) \\ & ([e, 2, e_1],\ add(\sigma, x, in_1),\ [v_1, c, 1]) \\ & ([e, 2, e_1, c, e_3],\ add(\sigma, x, in_1),\ [w_1]) \\ & ([e, 2, e_1],\ add(\sigma, x, in_3),\ [v_3, c, 1]) \\ & ([e, 2, e_1, c, e_3],\ add(\sigma, x, in_3),\ [w_3]) \} \end{aligned}$$

Finally, we complete $Prov(\varphi_1, R)$:

$$\begin{aligned} Prov(\varphi_1, R) = & \bigcup_{i \in \{1,3\}} [e, 2] \cdot Prov(\varphi_1', S_{1,i}) \cup \{([e], \sigma, \varphi_1)\} \cup \\ & \qquad \bigcup_{i \in \{1,3\}} \ \bigcup_{(\Phi, add(\sigma, x, in_i), \varphi) \in \mathbb{O}_{1,i}} [e, 1] \cdot Prov([in] \cdot \varphi, S). \end{aligned}$$

In the first two rows of Figure 4.3, we see how Rule P.18 maps φ_1' between the final result *out* of R, and the final result values v_1 and v_3 of, respectively, runs $S_{1,1}$ and $S_{1,3}$. In the last two rows, we see how Rule P.18 tracks the bound variable: for $i \in \{1, 3\}$, it maps $[in_i, b, 4]$ from the triple in $\mathbb{O}_{1,i}$ to the same path in in, the final result of run S. In rows 2–4, for $i \in \{1, 3\}$, we see that Figure 4.2 provides the connection between $[v_i, b, 4]$ and $[in_i, b, 4]$, with the missing part inserted here in dotted lines. Observe that $[in_i, b, 4]$ from Figure 4.2 occurs in triple $([e_1, b, e_2, x],\ add(\sigma, x, in_i),\ [in_i, b, 4])$ from $Prov(\varphi_1', S_{1,i})$, that also belongs to $\mathbb{O}_{1,i}$. From subexpression invocation $([e_1, b, e_2, x],\ add(\sigma, x, in_i))$ in run $S_{1,i}$, we know that $[in_i, b, 4]$ is a subvalue path of in_i.[††] From Rule P.2, for $i \in \{1, 3\}$,

$$Prov([in, in_i, b, 4], S) = \{([y], \sigma, [in, in_i, b, 4])\}.$$

Therefore,

$$Prov(\varphi_1, R) = \{ ([e],\ \sigma,\ [out, \langle b\colon 4, c\colon 1\rangle, b, 4])$$

[††]Here, $[in_1, b, 4]$ and $[in_3, b, 4]$ are convenient notations for, respectively, $[\langle a\colon 2, b\colon 4\rangle, b, 4]$ and $[\langle a\colon 5, b\colon 4\rangle, b, 4]$. In the run and provenance-triples, results and subvalue paths are simply values, without any identifiers we can refer to. To access the correct result value v in a run, we must use a subexpression invocation. The same subexpression invocation used in a provenance set gives access to a subvalue path of v.

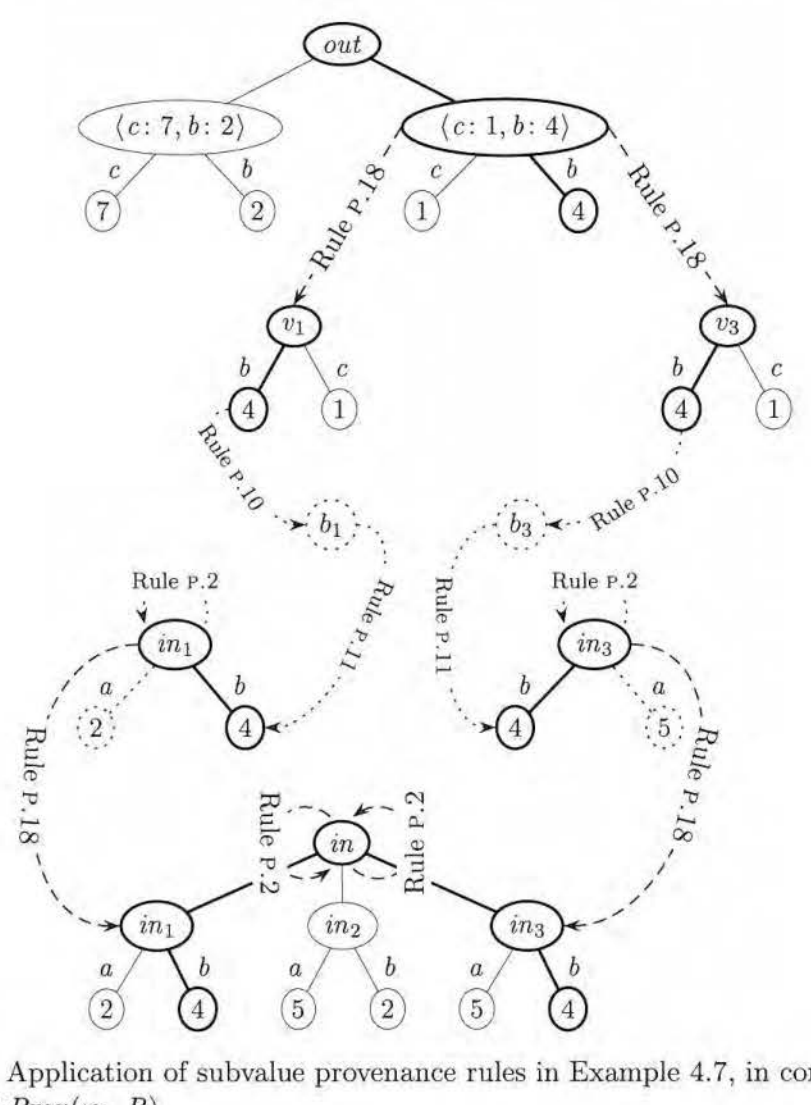


Figure 4.3: Application of subvalue provenance rules in Example 4.7, in computation of $\mathit{Prov}(\varphi_1, R)$.

$$
\begin{aligned}
&([e, 2, e_1],\ \mathit{add}(\sigma, x, in_1),\ [v_1, b, 4])\\
&([e, 2, e_1, b, e_2],\ \mathit{add}(\sigma, x, in_1),\ [b_1])\\
&([e, 2, e_1, b, e_2, x],\ \mathit{add}(\sigma, x, in_1),\ [in_1, b, 4])\\
&([e, 2, e_1],\ \mathit{add}(\sigma, x, in_3),\ [v_3, b, 4])\\
&([e, 2, e_1, b, e_2],\ \mathit{add}(\sigma, x, in_3),\ [b_3])\\
&([e, 2, e_1, b, e_2, x],\ \mathit{add}(\sigma, x, in_3),\ [in_3, b, 4])\\
&([e, 1, y],\ \sigma,\ [in, in_1, b, 4])
\end{aligned}
$$

$$([e,1,y],\ \sigma,\ [in, in_3, b, 4])\ \}$$

Notice the difference between the provenance sets $Prov(\varphi_1, R)$ and $Prov(\varphi_2, R)$. In $Prov(\varphi_1, R)$, we find triples that show us from which part of the input in, subvalue φ_1 was obtained. In $Prov(\varphi_2, R)$, we can only trace φ_2 as far as the result value of e_3, which is a service call. We cannot trace any further, because the service call is here a black box: we do not now how the result of e_3 depends on its input. □

4.2.3 Subvalue *where*-provenance

Let e be an NRC expression, σ a value assignment over $FV(e)$, and ζ a function assignment over $SN(e)$. Let $\sigma, \zeta \models\!\approx e \Rightarrow R$ and $v = result(R)$. Let φ be an occurrence of a subvalue of v.

Suppose we want to answer the question:

"Where does φ originate from in R?"

Assume that e only contains service-call expressions that represent external services. We have the following cases:

1. Subvalue φ originates from a constant in e.
2. Subvalue φ originates from a free variable in e.
3. Subvalue φ originates from an empty-set expression in e.
4. Subvalue φ originates from a result of an NRC expression in e that assembles its result from the results of its constituent subexpressions.
5. Subvalue φ originates from the result of an equality test or an emptiness test. These NRC expressions create their results. Indeed, for an equality test or an emptiness test, its result value is influenced by the values of the constituent subexpressions, but it is not assembled from those values. It is simply created by the expression.
6. Subvalue φ originates from the result of a service-call expression in e. For a service-call expression that represents an external service, its result value is provided by the called service, which is, in general, a black box in R. In this case we do not know anything about the relation between the output of the black box and its inputs, unless additional information is provided by the supplier of the service. Therefore, the tracking process stops.

The rules given in Section 4.2 clearly track the origin of φ in R as far as possible.

The tracking process stops at Rules P.1–P.3 for base NRC expressions, covering cases 1–3.

Most rules (P.4–P.11,P.14–P.18) apply to case 4. There, the tracking process continues into those subruns that have contributed to the assembling of φ. Note that in Rules P.14–P.15 for an if-expression, the tracking occurs only in the subrun corresponding to the actually executed branch of the expression. Indeed, although the final result of the subrun of the condition does influence the final result of R, φ cannot originate from that subrun. Note also that Rules P.16–P.18 take care of the tracking of subvalues of their bound variables.

Rule P.12 obviously applies to case 5 and also stops the tracking process.

Rule P.13 applies to case 6. As the service-call represents a black box, the tracking process stops.

Taking all this into consideration, we argue that repeated application of rules for computing subvalue provenance produces a suitable answer to the question, i.e., we can use the rules to compute the *where*-provenance of a subvalue, in the same sense as the concept of where-provenance defined in the context of databases[BKT01, CCT09]. We can either use the rules to compute the whole provenance set $Prov(\varphi, R)$, which can be queried further, or we can use the rules in an interactive way, as we illustrate in Section 4.4.

In the field of database theory, Cheney, Acar and Ahmed [CAA08] have defined a notion of "trace" of an NRC expression evaluation, which is similar to our notion of run. Moreover, they define an annotation propagation semantics for NRC, following Buneman, Cheney and Vansummeren [BCV08], over annotated inputs. An annotated input consists of an input σ and an annotation function h that maps each subvalue occurrence to some annotation value. For where-provenance, this function h can be taken to be the identity. The propagation semantics propagates annotations from the input into the result value of the NRC expression. Subvalues of the result value that have been constructed during the evaluation receive a dummy annotation $\perp$. They have shown that instead of evaluating an expression e using the annotation propagation semantics, on an input σ annotated by annotation function h, one can equivalently obtain the annotated result by propagating h through the trace of e on σ. They have formally proven that a subvalue in the final result has a non-$\perp$ annotation if and only if that subvalue was copied from the input.

Our definition of subvalue provenance works bottom-up and "backwards", rather than "forward" by propagation. It remains an open problem to formally reconcile these two approaches. We point out, however, that our definition of subvalue provenance allows tracking back the origin of a subvalue even if it was constructed during expression evaluation. In this way we go beyond merely extracting copies from the input.

As yet, we have assumed that all service-call expressions in e represent black boxes. This view conforms to the approach taken by the participants of the Provenance Challenges[‡‡], for the chosen example workflows. This view is consistent with the way their workflow systems operate. However, they had additional information about the behaviour of the black boxes, in the sense that all the black boxes were functional: the output of a black box depends on all of its inputs. Moreover, information was present that all inputs directly contributed to the output, i.e., the inputs were transformed into the output. Clearly, such information about black boxes is valuable. In Section 4.2.4 we investigate how we can use such information to augment the where-provenance of a subvalue.

Many workflow systems allow the reuse of existing dataflows as tasks in other dataflows. We have modelled this concept as service-call expressions that can be bound to subdataflows. Obviously, in this case, the service call is not a black box anymore, so we should be able to continue the tracking process into the corresponding run of the used subdataflow. Note that before executing e, we could simply substitute the subdataflow's expression in place of the service-call expression (with necessary changes to the function assignment). Then instead of a run of a subdataflow we simply have a subrun in R, and we can use the subvalue provenance function as defined. However, it is interesting to revisit the definition of subvalue provenance to take subdataflows explicitly into account, which we do in Section 4.3.

4.2.4 Turning a black box into a gray one, if possible

Thus far, we have treated service-call expressions bound to external services as impenetrable black boxes. Indeed, in general, we do not know any dependencies between the subvalues of the result of the call and subvalues of its actual parameters. The question is, whether we can whiten some of the black boxes. Missier et al. [MBZ$^+$08] have suggested using semantic rules to model dependencies between outputs and inputs of services.

In our model, we can include asserted dependencies in computing subvalue provenance.

[‡‡] `http://twiki.ipaw.info/bin/view/Challenge/WebHome`

Example 4.8. We continue to examine $Prov(\varphi_2, R)$ from Example 4.7. Since e_3 is a service call, in general, we cannot establish dependencies between its output and its input. Suppose, that we have additional information that the output of e_3 indeed directly depends, in a sense, on its input. In each execution of e_1, the subrun that produced the input of e_3 may also contribute to subvalue provenance. However, we cannot relate the subvalue of the result of e_3 to its input. We only can examine the subvalue provenance of the entire final result value of e_4 in the run of e_4.

For $i \in \{1, 3\}$, let $S_i = subrun([e, 2, e_1, c, e_3, 1, e_4], add(\sigma, x, in_i), R)$. Then

$$\begin{aligned}Prov([result(S_i)], S_i) = [e_4] \cdot Prov([in_i, a, result(S_i)], S_i') \\ \cup \{([e_4], add(\sigma, x, in_i), [result(S_i)])\}\end{aligned}$$

with $S_i' = subrun([e_4, x], add(\sigma, x, in_i), S_i)$, and

$$Prov([in_i, a, result(S_i)], S_i') = \{([x], add(\sigma, x, in_i), [in_i, a, result(S_i')])\}$$

We could thus consider incorporating the following set into $Prov(\varphi_2, R)$:

$$\begin{aligned}\{\, &([e, 2, e_1, c, e_3, 1, e_4],\ add(\sigma, x, in_1),\ [a_1]) \\ &([e, 2, e_1, c, e_3, 1, e_4, x],\ add(\sigma, x, in_1),\ [in_1, a, 2]) \\ &([e, 2, e_1, c, e_3, 1, e_4],\ add(\sigma, x, in_3),\ [a_3]) \\ &([e, 2, e_1, c, e_3, 1, e_4, x],\ add(\sigma, x, in_3),\ [in_3, a, 5])\,\}\end{aligned}$$

The set, however, is not yet complete. Remember that e_4 was executed inside a for-expression, therefore we should further track the bound variable x. For $i \in \{1, 3\}$,

$$Prov([in, in_i, a, result(S_i)], S) = \{([e, 1, y], \sigma, [in, in_i, a, result(S_i)])\}$$

Thus we could also consider incorporating the following set into $Prov(\varphi_2, R)$:

$$\begin{aligned}\{\, &([e, 1, y],\ \sigma,\ [in, in_1, a, 2]) \\ &([e, 1, y],\ \sigma,\ [in, in_3, a, 5])\,\}\end{aligned}$$

However, adding these sets to $Prov(\varphi_2, R)$ alters the semantics of the provenance set, as discussed in Section 4.2.3. Indeed, the additional triples do not contribute to the where-provenance of φ_2 in R. But they do show the where-provenance of the inputs of e_3, on which the final results of e_3 somehow depend. □

In general, we cannot even assume that for a service call with one input, its output is directly computed from the input. Indeed, that service call might invoke an external service that uses the input to select records in a database query, and then computes the output from the query result. Each service call, in principle, might have side effects.

Therefore, we define a *dependency assignment*, that can be specified, either before or after executing a dataflow, for a given function assignment.

Definition 4.9. Let $N \subseteq \mathcal{N}$. Let ζ be a function assignment over N. We define a *dependency assignment* Dep_ζ as a mapping from N to $\mathbb{P}^{\text{fin}}(\mathbb{N})$, such that the following holds:

$$\forall f \in N \colon (\zeta(f) \colon [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!] \to [\![\tau_{out}]\!] \implies Dep_\zeta(f) \subseteq \{1, \ldots, n\}) .$$

These natural numbers positionally identify the inputs of $\zeta(f)$ on which a result value of $\zeta(f)$ *directly depends.*

We do not define what *directly depends* exactly means, other that the dependency is sufficient for the designer of the dataflow to assert its existence in Dep_ζ. Additional annotations may be used to explain the significance of the dependencies in Dep_ζ.

Note 4.2.1. To illustrate different understandings of what it means that an output of a service call directly depends on its input, consider some external services *filter₁* and *filter₂*, that both take one parameter *criterion*. Suppose *filter₁* executes the left-hand side SQL query, and *filter₂* the other one:

```
SELECT a, b, c            SELECT criterion AS criterion, a, b, c
  FROM Table                FROM Table
 WHERE f(a) = criterion    WHERE f(a) = criterion
```

On one hand, we could consider the output of *filter₂* to directly depend on *criterion*, as the parameter is actually used in the construction of the output. In this sense, the output of *filter₁* does not directly depend on *criterion*.

On the other hand, we could decide that the fact that *criterion* is used to filer records is already strong enough to assert a dependency. In this case, the output of both *filter₁* and *filter₂* directly depends on *criterion*.

One could even consider a dependence between *criterion* and the output of the following query:

```
SELECT a, b, c, g(criterion) as d
  FROM Table
 WHERE f(a) = criterion
```

□

What we do assume, is the following notion of *independence*.

Definition 4.10. Let f be a non-deterministic mapping from $X_1 \times \cdots \times X_n$ to Y. We say that f is *independent* of the i-th input if

$$\begin{aligned}&\forall\, (v_1, \ldots, v_{i-1}, v_{i+1}, \ldots v_n) \in X_1 \times \cdots \times X_{i-1} \times X_{i+1} \times \cdots \times X_n \\ &\quad \forall\, w_i, w_i' \in X_i \colon (v_1, \ldots, w_i, \ldots, v_n) \in [\![f]\!] \iff (v_1, \ldots, w_i', \ldots, v_n) \in [\![f]\!]\,.\end{aligned}$$

We can now alter the definition of subvalue provenance to take such information into account.

Let e be an NRC expression, and σ a value assignment over $FV(e)$. Let ζ be a function assignment over $SN(e)$, and Dep_ζ a dependency assignment for ζ. Let $\sigma, \zeta \approx\!\models e \Rightarrow R$ and $v = result(R)$. Let φ be an occurrence of a subvalue of v, i.e., $\varphi \leftarrow\!\!\bullet\, v$. For each $f \in SN(e)$, let f be independent of all inputs in positions not included in $Dep_\zeta(f)$.

We define the *asserted subvalue provenance function*, denoted by $Gray()$, by adjusting the Rules P.1–P.18 as follows:

- for every rule except Rule P.13, we simply use the same rule with every occurrence of $Prov()$ replaced by $Gray()$.
- we replace Rule P.13 with the following one:

$$\frac{\begin{array}{c} e = f(e_1, \ldots, e_n) \qquad \sigma, \zeta \approx\!\models e \Rightarrow R \\ \varphi \leftarrow\!\!\bullet\, result(R) \qquad \forall\, i \in Dep_\zeta(f) \colon S_i = subrun([e, i, e_1], \sigma, R) \end{array}}{Gray(\varphi, R) \overset{def}{=} \bigcup_{i \in Dep_\zeta(f)} [e] \cdot Gray([result(S_i)], S_i) \,\cup\, \{([e],\, \sigma,\, \varphi)\}} \quad \text{G.13}$$

We call $Gray(\varphi, R)$ "the asserted subvalue provenance of φ in R".

Rule G.13 essentially adds the subvalue provenance of the relevant inputs of the service call, in their corresponding subruns. Note that it corresponds to the informal notion of "contribution" in the context of workflows, where provenance information includes subvalues from the input that were transformed into v by a computation.

We should mention that the Open Provenance Model [MCF+11] provides means to explicitly model dependencies between data in a run. A run is modelled as a graph, in which data is represented as artifact-nodes. The dependency between artifact-nodes is expressed by the so called "was-derived-from edges". Currently, the artifact-nodes in OPM are treated as atomic data, although the need for support of complex data has been acknowledged in the OPM community. Recently, Anand et al. [ABAL10] have presented a graph model that accommodates structured data, modelled by XML trees, recording dependency relations between nodes in the trees. Also Chapman and Jagadish [CJ10] have tackled the problem of black boxes, and have proposed a model for recording dependencies provided by a black box. It would be interesting to investigate whether we can, for some classes of black boxes, express the dependencies between subvalues of the inputs and subvalues of the output in the form of provenance rules.

4.3 Subvalue provenance in the context of a dataflow repository

Recall the formal model of a dataflow repository we have defined in Section 3.3. Let $r \in \mathcal{R}$ be a run identifier and let $R = run(r)$. Let $e_r = expr(dataflow(r))$, $\sigma_r = values(r)$, and $\zeta_r = \zeta_{binding(r)}$. Let φ_r be an occurrence of a subvalue of $result(R)$, i.e., $\varphi_r \multimapinv result(R)$.

Remember that we represent the provenance set $Prov(\varphi_r, R)$ as a set consisting of triples. Each triple is of the form $(\Phi', \sigma', \varphi')$, where, Φ' is an occurrence of subexpression e' of e_r, with $\sigma', \zeta_r \models e' \Rightarrow v'$, and where φ' is an occurrence of a subvalue of v', i.e., $\varphi' \multimapinv v'$. We also know that $(\Phi', \sigma') \in SI(R)$ and $v' = result(subrun(\Phi', \sigma', R))$ (Proposition 4.4). The definition of $Prov()$ operates only in the context of a given run, and thus provides only the desired result, being the where-provenance of a subvalue, for runs of expressions whose service-call subexpressions represent external services.

In this section, we redefine the subvalue provenance function in the context of a dataflow repository, so that service-call expressions representing subdataflows can be handled properly. Indeed, if a service-call expression is bound to a subdataflow, the tracking of subvalue provenance should continue into the correct run of that subdataflow. As the subdataflow may contain service-call expressions as well, the tracking process can be stopped if the service call is a black box. However, if it is a subsubdataflow, the tracking process can continue, and so forth. If the tracking process, while in the run of a subdataflow, does not encounter any black boxes, than the tracking process can return to the run of the parent dataflow.

We represent the *extended subvalue provenance* P^* *of* φ_r *in* $run(r)$ as a set consisting of pairs. Each pair is of the form $(r', (\Phi', \sigma', \varphi'))$, where $(\Phi', \sigma') \in SI(run(r'))$, and $\varphi' \longleftarrow\!\!\bullet\; result(subrun(\Phi', \sigma', run(r')))$. Essentially, extended subvalue provenance "tags" provenance-triples with the identifier of the run in which the provenance was being tracked.

In Rules E.1–E.18, we define the *extended subvalue provenance function*, denoted by $Prov^*()$. In general, we refer to sets produced by $Prov^*()$ as *extended provenance sets.* We refer to $Prov^*(\varphi_r, r \parallel [e_r], \sigma_r)$ as *the extended subvalue provenance of* φ_r *in* $run(r)$.

In defining $Prov^*()$, we use a slightly different notation. Instead of passing a whole run as an argument, we can use its run identifier and a subexpression invocation (the two arguments after the "$\|$" symbol) to identify the correct subrun, in which the tracking process should continue. Each rule is thus aware of the full subexpression path, and can use it in its tagged provenance-triple.

Rules E.1–E.12 and E.14–E.18 are a tedious but straightforward adaptation of their corresponding rules from Section 4.2.2.

$$\frac{r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \qquad \Phi = \Phi' \cdot [e] \qquad e = \mathtt{a}}{Prov^*([\mathtt{a}], r \parallel \Phi, \sigma) \overset{def}{=} \{(r, (\Phi,\ \sigma,\ [\mathtt{a}]))\}} \text{ E.1}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \\ \Phi = \Phi' \cdot [e] \qquad e = x \qquad \varphi \longleftarrow\!\!\bullet\; result(subrun(\Phi, \sigma, run(r))) \end{array}}{Prov^*(\varphi, r \parallel \Phi, \sigma) \overset{def}{=} \{(r, (\Phi,\ \sigma,\ \varphi))\}} \text{ E.2}$$

$$\frac{r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \qquad \Phi = \Phi' \cdot [e] \qquad e = \varnothing}{Prov^*([\emptyset], r \parallel \Phi, \sigma) \overset{def}{=} \{(r, (\Phi,\ \sigma,\ [\emptyset]))\}} \text{ E.3}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \qquad \Phi = \Phi' \cdot [e] \\ e = \{e'\} \text{ or } e = \bigcup e' \qquad v := result(subrun(\Phi, \sigma, run(r))) \\ S := subrun(\Phi \cdot [e'], \sigma, run(r)) \qquad \varphi' = [result(S)] \end{array}}{Prov^*([v], r \parallel \Phi, \sigma) \overset{def}{=} Prov^*(\varphi', r \parallel \Phi \cdot [e'], \sigma) \cup \{(r, (\Phi,\ \sigma,\ [v]))\}} \text{ E.4}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \qquad \Phi = \Phi' \cdot [e] \qquad e = \{e'\} \\ S := subrun(\Phi, \sigma, run(r)) \qquad \varphi \mathrel{\leftarrow\!\!\dot{\bullet}} result(S) \qquad \varphi = [result(S)] \cdot \varphi' \end{array}}{Prov^*(\varphi, r \parallel \Phi, \sigma) \overset{def}{=} Prov^*(\varphi', r \parallel \Phi \cdot [e'], \sigma) \cup \{(r, (\Phi,\ \sigma,\ \varphi))\}} \ \text{E.5}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \\ \Phi = \Phi' \cdot [e] \qquad e = \bigcup e' \qquad S := subrun(\Phi, \sigma, run(r)) \\ \varphi \mathrel{\leftarrow\!\!\dot{\bullet}} result(S) \qquad \varphi = [result(S)] \cdot \varphi' \qquad \varphi' \mathrel{\leftarrow\!\!\bullet} u \\ S' := subrun(\Phi \cdot [e'], \sigma, run(r)) \qquad W := \{w \in result(S') \mid u \in w\} \end{array}}{\begin{array}{l} Prov^*(\varphi, r \parallel \Phi, \sigma) \overset{def}{=} \{(r, (\Phi,\ \sigma,\ \varphi))\} \cup \\ \qquad\qquad \displaystyle\bigcup_{w \in W} Prov^*([result(S'), w] \cdot \varphi', r \parallel \Phi \cdot e', \sigma) \end{array}} \ \text{E.6}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \qquad \Phi = \Phi' \cdot [e] \\ e = (e_1 \cup e_2) \qquad S := subrun(\Phi, \sigma, run(r)) \qquad v := result(S) \\ \forall i \in \{1.2\} \colon S_i := subrun(\Phi \cdot [i, e_i], \sigma, run(r)) \wedge \varphi_i = [result(S_i)] \end{array}}{Prov^*([v], r \parallel \Phi, \sigma) \overset{def}{=} \displaystyle\bigcup_{i \in \{1,2\}} Prov^*(\varphi_i, r \parallel \Phi \cdot [i, e_i], \sigma) \cup \{(r, (\Phi,\ \sigma,\ [v]))\}} \ \text{E.7}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \qquad \Phi = \Phi' \cdot [e] \qquad e = (e_1 \cup e_2) \\ S := subrun(\Phi, \sigma, run(r)) \qquad \varphi \mathrel{\leftarrow\!\!\dot{\bullet}} result(S) \qquad \varphi = [result(S)] \cdot \varphi' \\ \forall i \in \{1, 2\} \colon S_i := subrun(\Phi \cdot [i, e_i], \sigma, run(r)) \wedge v_i := result(S_i) \\ I := \{i \in \{1, 2\} \mid [v_i] \cdot \varphi' \mathrel{\leftarrow\!\!\bullet} v_i\} \end{array}}{Prov^*(\varphi, r \parallel \Phi, \sigma) \overset{def}{=} \displaystyle\bigcup_{i \in I} Prov^*([v_i] \cdot \varphi', r \parallel \Phi \cdot [i, e_i], \sigma) \cup \{(r, (\Phi,\ \sigma,\ \varphi))\}} \ \text{E.8}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \\ \Phi = \Phi' \cdot [e] \qquad e = \langle l_1 \colon e_1, \ldots, l_n \colon e_n \rangle \qquad S := subrun(\Phi, \sigma, run(r)) \\ v := result(S) \qquad \forall i \in \{1, \ldots, n\} \colon S_i := subrun(\Phi \cdot [l_i, e_i], \sigma, run(r)) \end{array}}{\begin{array}{l} Prov^*([v], r \parallel \Phi, \sigma) \overset{def}{=} \{(r, (\Phi,\ \sigma,\ [v]))\} \cup \\ \qquad\qquad \displaystyle\bigcup_{i \in \{1, \ldots, n\}} Prov^*([result(S_i)], r \parallel \Phi \cdot [l_i, e_i], \sigma) \end{array}} \ \text{E.9}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \\ (\Phi, \sigma) \in SI(run(r)) \qquad \Phi = \Phi' \cdot [e] \qquad e = \langle l_1 \colon e_1, \ldots, l_n \colon e_n \rangle \\ S := subrun(\Phi, \sigma, run(r)) \qquad \varphi \leftarrow\!\dot{\bullet}\; result(S) \\ \varphi = [result(S), l_i] \cdot \varphi' \qquad S' := subrun(\Phi \cdot [l_i, e_i], \sigma, run(r)) \end{array}}{Prov^*(\varphi, r \parallel \Phi, \sigma) \overset{def}{=} Prov^*(\varphi', r \parallel \Phi \cdot [l_i, e_i], \sigma) \cup \{(r, (\Phi,\ \sigma,\ \varphi))\}} \;\text{E.10}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \\ \Phi = \Phi' \cdot [e] \qquad e = e'.l \qquad S := subrun(\Phi, \sigma, run(r)) \\ \varphi \leftarrow\!\bullet\; result(S) \qquad S' := subrun(\Phi \cdot [e'], \sigma, run(r)) \end{array}}{\begin{array}{l} Prov^*(\varphi, r \parallel \Phi, \sigma) \overset{def}{=} \{(r, (\Phi,\ \sigma,\ \varphi))\} \cup \\ \qquad\qquad\qquad Prov^*([result(S'), l] \cdot \varphi, r \parallel \Phi \cdot [e'], \sigma) \end{array}} \;\text{E.11}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \\ (\Phi, \sigma) \in SI(run(r)) \qquad \Phi = \Phi' \cdot [e] \qquad e = (e_1 = e_2) \text{ or } e = (e' = \varnothing) \\ S := subrun(\Phi, \sigma, run(r)) \qquad v := result(S) \end{array}}{Prov^*([v], r \parallel \Phi, \sigma) \overset{def}{=} \{(r, (\Phi,\ \sigma,\ [v]))\}} \;\text{E.12}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \\ \Phi = \Phi' \cdot [e] \qquad e = \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \qquad S := subrun(\Phi, \sigma, run(r)) \\ \varphi \leftarrow\!\bullet\; result(S) \qquad result(subrun(\Phi \cdot [0, e_0], \sigma, run(r))) = \mathtt{true} \end{array}}{Prov^*(\varphi, r \parallel \Phi, \sigma) \overset{def}{=} Prov^*(\varphi, r \parallel \Phi \cdot [1, e_1], \sigma) \cup \{(r, (\Phi,\ \sigma,\ \varphi))\}} \;\text{E.14}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \\ \Phi = \Phi' \cdot [e] \qquad e = \mathsf{if}\ e_0\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \qquad S := subrun(\Phi, \sigma, run(r)) \\ \varphi \leftarrow\!\bullet\; result(S) \qquad result(subrun(\Phi \cdot [0, e_0], \sigma, run(r))) = \mathtt{false} \end{array}}{Prov^*(\varphi, r \parallel \Phi, \sigma) \overset{def}{=} Prov^*(\varphi, r \parallel \Phi \cdot [2, e_2], \sigma) \cup \{(r, (\Phi,\ \sigma,\ \varphi))\}} \;\text{E.15}$$

We also need to adapt the definition for the function selecting provenance-triples with a subexpression path ending in a particular variable.

Definition 4.11. Let U, X, Y, and Z be sets. For $P \subseteq U \times (SEQ(X) \times Y \times Z)$, for $u \in U$, $x \in X$, and $y \in Y$, we define:

$$Sel(u, x, y, P) \overset{def}{=} \{(u, (q, y, z)) \in P \mid x \text{ is the } \textit{last} \text{ element of } q\}\ .$$

Now we can adapt the rules for the let- and for-expressions.

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \qquad \Phi = \Phi' \cdot [e] \\ e = \mathsf{let}\ x := e_1\ \mathsf{in}\ e_2 \qquad S := subrun(\Phi, \sigma, run(r)) \qquad \varphi \leftarrow\!\!\bullet\ result(S) \\ S_1 := subrun(\Phi \cdot [1, e_1], \sigma, run(r)) \qquad \sigma' := add(\sigma, x, result(S_1)) \\ P^* := Prov^*(\varphi, r \parallel \Phi \cdot [2, e_2], \sigma') \qquad \mathbb{O} := Sel(r, x, \sigma', P^*) \end{array}}{\begin{array}{l} Prov^*(\varphi, r \parallel \Phi, \sigma) \overset{def}{=} P^* \cup \{(r, (\Phi,\ \sigma,\ \varphi))\} \cup \\ \qquad\qquad \bigcup\limits_{(r,(\Phi'',\sigma',\varphi')) \in \mathbb{O}} Prov^*(\varphi', r \parallel \Phi \cdot [1, e_1], \sigma) \end{array}} \text{ E.16}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi, \sigma) \in SI(run(r)) \\ \Phi = \Phi' \cdot [e] \qquad e = \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2 \qquad S := subrun(\Phi, \sigma, run(r)) \\ \varphi = [result(S)] \qquad S_1 := subrun(\Phi \cdot [1, e_1], \sigma, run(r)) \\ \forall w \in result(S_1)\colon \sigma_w := add(\sigma, x, w) \\ \forall w \in result(S_1)\colon S_w := subrun(\Phi \cdot [2, e_2], \sigma_w, run(r)) \\ \forall w \in result(S_1)\colon P^*_w := Prov^*([result(S_w)], r \parallel \Phi \cdot [2, e_2], \sigma_w) \\ \forall w \in result(S_1)\colon \mathbb{O}_w := Sel(r, x, \sigma_w, P^*_w) \end{array}}{\begin{array}{l} Prov^*(\varphi, r \parallel \Phi, \sigma) \overset{def}{=} \bigcup\limits_{w \in result(S_1)} P^*_w \cup \{(r, (\Phi,\ \sigma,\ \varphi))\} \cup \\ \quad \bigcup\limits_{w \in result(S_1)} \ \bigcup\limits_{(r,(\Phi'',\sigma_w,\varphi)) \in \mathbb{O}_w} Prov^*([result(S_1)] \cdot \varphi, r \parallel \Phi \cdot [1, e_1], \sigma) \end{array}} \text{ E.17}$$

$$\frac{\begin{array}{c} r \in \mathcal{R} \\ (\Phi, \sigma) \in SI(run(r)) \qquad \Phi = \Phi' \cdot [e] \qquad e = \mathsf{for}\ x\ \mathsf{in}\ e_1\ \mathsf{return}\ e_2 \\ S := subrun(\Phi, \sigma, run(r)) \qquad \varphi \leftarrow\!\!\dot{\bullet}\ result(S) \\ \varphi = [result(S)] \cdot \varphi' \qquad \varphi' \leftarrow\!\!\bullet\ u \qquad S_1 := subrun(\Phi \cdot [1, e_1], \sigma, run(r)) \\ \forall w \in result(S_1)\colon \sigma_w := add(\sigma, x, w) \\ \forall w \in result(S_1)\colon S_w := subrun(\Phi \cdot [2, e_2], \sigma_w, run(r)) \\ W := \{w \in result(S_1) \mid result(S_w) = u\} \\ \forall w \in W\colon P^*_w = Prov^*(\varphi', r \parallel \Phi \cdot [2, e_2], \sigma_w) \wedge \mathbb{O}_w := Sel(r, x, \sigma_w, P^*_w) \end{array}}{\begin{array}{l} Prov^*(\varphi, r \parallel \Phi, \sigma) \overset{def}{=} \bigcup\limits_{w \in W} P^*_w \cup \{(r, (\Phi,\ \sigma,\ \varphi))\} \cup \\ \quad \bigcup\limits_{w \in W} \ \bigcup\limits_{(r,(\Phi'',\sigma_w,\varphi'')) \in \mathbb{O}_w} Prov^*([result(S_1)] \cdot \varphi'', r \parallel \Phi \cdot [1, e_1], \sigma) \end{array}} \text{ E.18}$$

The last two rules deal with service-call expressions. The first rule is for a service-call expression that is bound to a subdataflow. The binding is extracted from the binding tree that is associated with r: there must be a child-node c of the root that is labelled with a dataflow identifier, and the connecting edge must be labelled with the service name from this service-call expression. If it is the case, the run identifier of the correct run of the sub-dataflow is provided by the mapping *internalCall*(). We then compute the extended subvalue provenance set P^*, in the correct run of the subdataflow. It is possible that the tracking process does not encounter any black boxes, and tracks up to the free variables of the subdataflow. We look for all such pairs and collect them in $\mathbb{O}$. For each of these pairs, we compute the extended subvalue provenance in the corresponding subrun of the parent dataflow, for the subvalue of the input that was tracked in P^*. To determine the correct subrun we use the replacement mapping from the binding tree.

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi,\sigma) \in SI(run(r)) \qquad \Phi = \Phi' \cdot [e] \qquad e = f(e_1,\dots,e_n) \\ \beta := binding(r) \qquad \beta = (T,\lambda,M) \,\wedge\, T = (r,V,E) \\ \exists\, c \in V\colon \lambda(r,c) = f \,\wedge\, \lambda(c) \in \mathcal{D} \qquad S := subrun(\Phi,\sigma,run(r)) \\ \varphi \multimap result(S) \qquad r' = internalCall(r,(\Phi,\sigma,result(S))) \\ P^* := Prov^*(\varphi, r' \parallel [expr(dataflow(r'))], values(r')) \\ \mathbb{O} := \{\, Sel(r',x,values(r'),P^*) \mid x \in FV(expr(dataflow(r')))\} \end{array}}{\begin{array}{l} Prov^*(\varphi, r \parallel \Phi,\sigma) \overset{def}{=} \{(r,(\Phi,\ \sigma,\ \varphi))\} \,\cup\, P^* \,\cup \\ \qquad \displaystyle\bigcup_{(r',(\Phi'',values(r'),\varphi'))\in\mathbb{O}} Prov^*(\varphi', r \parallel \Phi \cdot [M(c)(x), e_{M(c)(x)}], \sigma) \end{array}} \; \text{E.13S}$$

If the service-call expression is bound to an external service identifier, the tracking process stops.

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi,\sigma) \in SI(run(r)) \\ \Phi = \Phi' \cdot [e] \qquad e = f(e_1,\dots,e_n) \qquad \beta := binding(r) \\ \beta = (T,\lambda,M) \,\wedge\, T = (r,V,E) \qquad \exists\, c \in V\colon \lambda(r,c) = f \,\wedge\, \lambda(c) \in \mathcal{E} \\ S := subrun(\Phi,\sigma,run(r)) \qquad \varphi \multimap result(S) \end{array}}{Prov^*(\varphi, r \parallel \Phi,\sigma) \overset{def}{=} \{(r,(\Phi,\ \sigma,\ \varphi))\}} \; \text{E.13E}$$

Note that we can alternatively define the last rule to accommodate any asserted dependencies, as described in Section 4.2.4. In order to do so, we need to extend the dataflow repository with a mapping *asserted*() from $\mathcal{R}$ to $\mathcal{DA}$, with $\mathcal{DA}$ the set of all dependency assignments. We also need to add the following constraint: *asserted*(r) is a dependency assignment for $\zeta_{binding(r)}$, such that for each $f \in SN(expr(dataflow(r)))$, f is independent of all inputs in positions not included in *asserted*$(r)(f)$.

$$\frac{\begin{array}{c} r \in \mathcal{R} \qquad (\Phi,\sigma) \in SI(run(r)) \\ \Phi = \Phi' \cdot [e] \qquad e = f(e_1,\ldots,e_n) \qquad \beta := binding(r) \\ \beta = (T,\lambda,M) \,\wedge\, T = (r,V,E) \qquad \exists\, c \in V\colon \lambda(r,c) = f \,\wedge\, \lambda(c) \in \mathcal{E} \\ S := subrun(\Phi,\sigma,run(r)) \qquad \varphi \leftarrow\!\bullet\, result(S) \\ \forall\, i \in asserted(r)(f)\colon S_i = subrun(\Phi\cdot[i,e_i],\sigma,run(r)) \end{array}}{\begin{array}{l} Prov^*(\varphi, r \parallel \Phi,\sigma) \overset{def}{=} \{(r,(\Phi,\ \sigma,\ \varphi))\} \,\cup \\ \qquad\qquad \displaystyle\bigcup_{i\in asserted(r)(f)} Prov^*([result(S_i)], r \parallel \Phi\cdot[i,e_i],\sigma) \end{array}} \ \text{E.13G}$$

Observe the difference with Rule E.13S. Here, we can only compute the extended subvalue provenance of an entire input in its corresponding subrun.

4.4 Exploring a past execution

An interesting way of using our provenance inference rules is, instead of computing a whole provenance set at once, tracking subvalue provenance in a step-by-step manner. In the following, we use an example to show how we can use the rules for the extended subvalue provenance to explore a run within the context provided by the rules. Indeed, each rule already determines a selection of run-triples that contribute to subvalue provenance. We can choose to either follow the entire selection, or, in rules for a for-expression or a flatten, only a part of it. We can also choose to stop and "change direction", for example to explore the part of the run that has produced a boolean in a test or in a conditional.

Recall Example 3.3 (p. 42), where we have computed a run R for

$$e = \mathsf{for}\ x\ \mathsf{in}\ y\ \mathsf{return}\ \langle b\colon x.b, c\colon f(x.a)\rangle\,,$$

with $e_1 = \langle b\colon e_2, c\colon e_3\rangle$, $e_2 = x.b$, $e_3 = f(e_4)$, and $e_4 = x.a$. In Example 4.7 (p. 87), we have computed $Prov(\varphi_2, R)$ for $\varphi_2 = [out, \langle b\colon 4, c\colon 1\rangle, c, 1]$.

Now suppose that R is stored in a dataflow repository (conform to the model in Section 3.3), with run identifier r, i.e., $run(r) = R$ and $values(r) = \sigma$. Let $d = dataflow(r)$, then $expr(d) = e$. Suppose that we are still interested in subvalue φ_2 of the final result value of R, shown in the first tree of Figure 4.4.

We have tagged *out* with the run identifier r, to indicate to which run the value belongs. Instead of computing the whole $Prov^*(\varphi_2, r \parallel [e], \sigma)$, we redo the exercise by stepping through the rules. In the first step, we must use

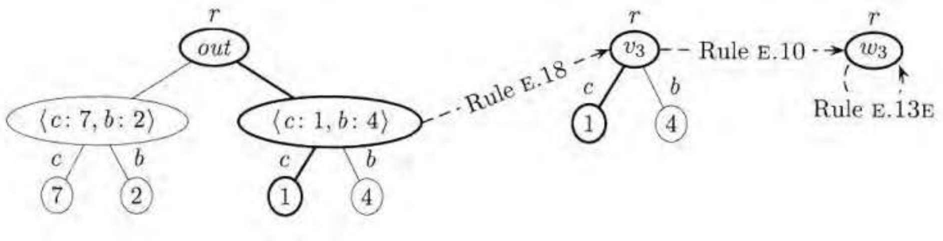


Figure 4.4: Subvalue provenance in partial computation of $Prov^*(\varphi_2, r \parallel [e], \sigma)$

Rule E.18 for a for-expression. The rule already produces the first tagged triple:

$$(r, ([e],\ \sigma,\ [out, \langle b\colon 4, c\colon 1\rangle, c, 1]))\,.$$

As in Example 4.7, we know $W = \{in_1, in_3\}$. The rule determines which subexpression invocations in R have to be considered in the second step, but we can choose, for instance, to only follow the branch for in_3. The rule also determines which subvalue of v_3 from the run-triple $([e, 2, e_1],\ add(\sigma, x, in_3),\ v_3)$ is going to be tracked, as shown in Figure 4.4.

Note that if we also want to trace the bound variable, we must compute the whole $Prov^*(\varphi_2', r \parallel [e, 2, e_1], add(\sigma, x, in_3))$.

In the second step, we must use Rule E.10 for a tuple-expression. This rule produces the second tagged triple:

$$(r, ([e, 2, e_1],\ add(\sigma, x, in_3),\ [v_3, c, 1]))\,,$$

and determines that next we track [1] from the run-triple

$$t = ([e, 2, e_1, c, e_3],\ add(\sigma, x, in_3),\ w_3)\,. \tag{4.1}$$

In the third step, we must use a rule for a service-call expression. In Example 4.7, e_3 represented an external service. That means that in $binding(r)$, there is an edge labelled f from the root to some node y, and that node is labelled with an external service identifier. We use Rule E.13E, which adds the last tagged triple:

$$t_p = (r, ([e, 2, e_1, c, e_3],\ add(\sigma, x, in_3),\ [w_3]))\,, \tag{4.2}$$

and stops the tracking process. That also means the we have the whole $Prov^*(\varphi_2', r \parallel [e, 2, e_1], add(\sigma, x, in_3))$:

$$\begin{aligned}\{\,&(r, ([e, 2, e_1],\ add(\sigma, x, in_3),\ [v_3, c, 1]))\\ &(r, ([e, 2, e_1, c, e_3],\ add(\sigma, x, in_3),\ [w_3]))\,\}\,.\end{aligned}$$

As none of the subexpression paths ends in the bound variable x of the for-expression, we cannot track any further, so we have reached the run-triple from *where* φ_2 originates.

Note that at this point, we can also choose to use Rule E.13G instead of Rule E.13E, if we know that the output of f depends on its input, and either compute

$$P^* = \mathit{Prov}^*([a_3], r \parallel [e, 2, e_1, c, e_3, 1, e_4], \mathit{add}(\sigma, x, \mathit{in}_3))\,,$$

or again, step through it. From Example 4.8 (p. 94) we already know P^* contains a tagged triple with a subexpression path ending in x, which can be used to track further till we reach the input value in.

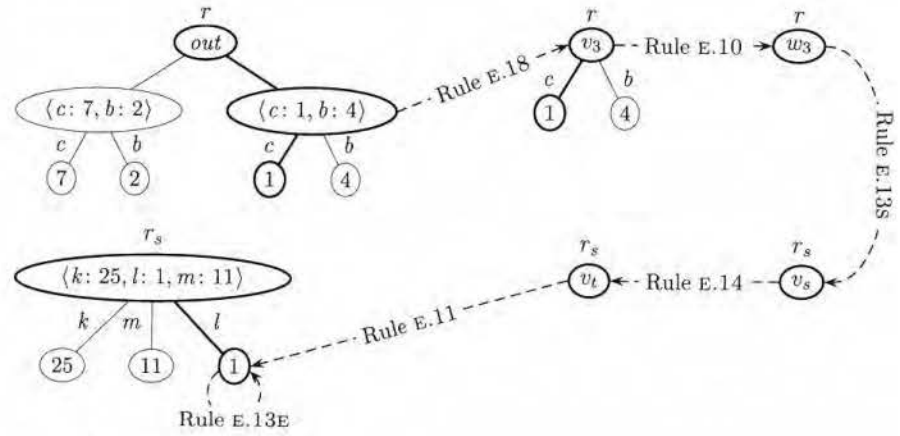
Observe that if we reach an equality-test, an emptiness-test, or an if-expression, we can, in the same way, explore the part of the run that produced the boolean result value. Moreover, instead of a subvalue of out, we can also track any subvalue of any intermediate result of R. Indeed, for a triple $(\Phi', \sigma', v') \in R$, and $\varphi' \multimap v'$, we can either compute $\mathit{Prov}^*(\varphi', r \parallel \Phi', \sigma')$, or step through its computation.

To show how the extended subvalue provenance deals with subdataflows, we need to slightly modify the example. Suppose that in the third step, e_3 represents a subdataflow with identifier d_s. That means that in $\mathit{binding}(r)$, there is an edge labelled f from the root to some node y labelled with d_s. Suppose $\mathit{expr}(d_s)$ is the following expression:

$$\mathsf{if}\ x = 5\ \mathsf{then}\ g(x).l\ \mathsf{else}\ 0\,,$$

with $e_s = \mathsf{if}\ e_c\ \mathsf{then}\ e_t\ \mathsf{else}\ 0$, $e_c = (x = 5)$, $e_t = e_g.l$, and $e_g = g(x)$. Suppose g is bound to an external service. The identifier of the relevant run of d_s is $\mathit{internalCall}(r, t) = r_s$ (Eq. 4.1). Let $\sigma_3 = \mathit{add}(\sigma, x, \mathit{in}_3)$, then $\mathit{values}(r_s) = \mathit{add}(\sigma_3, x, 5)$, assuming the replacement mapping for f maps the free variable x of e_s to the only input of f. Suppose $\mathit{run}(r_s)$ is the following set:

$$\begin{aligned}
\{\, &([e_s, 0, e_c, 1, x],\ \mathit{add}(\sigma_3, x, 5),\ v_x),\\
&([e_s, 0, e_c, 2, 5],\ \mathit{add}(\sigma_3, x, 5),\ v_5),\\
&([e_s, 0, e_c],\ \mathit{add}(\sigma_3, x, 5),\ \mathtt{true}),\\
&([e_s, 1, e_t, e_g, 1, x],\ \mathit{add}(\sigma_3, x, 5),\ v_g),\\
&([e_s, 1, e_t, e_g],\ \mathit{add}(\sigma_3, x, 5),\ \langle k\colon 25, l\colon 1, m\colon 11\rangle),\\
&([e_s, 1, e_t],\ \mathit{add}(\sigma_3, x, 5),\ v_t),\\
&([e_s],\ \mathit{add}(\sigma_3, x, 5),\ v_s)\,\}
\end{aligned}$$



r
out
$\langle c\colon 7, b\colon 2\rangle$
$\langle c\colon 1, b\colon 4\rangle$
c
b
7
2
1
4
Rule E.18
v_3
Rule E.10
w_3
Rule E.13S
r_s
$\langle k\colon 25, l\colon 1, m\colon 11\rangle$
k
m
l
25
11
1
v_t
Rule E.14
v_s
Rule E.11
Rule E.13E

Figure 4.5: Subvalue provenance in a run of a subdataflow

with $v_s = v_t = 1$ and $v_x = v_5 = v_g = 5$. Instead of Rule E.13E, we now use Rule E.13S, which also adds the tagged triple t_p (Eq. 4.2). We can track further, by stepping through the computation of $Prov^*([1], r_s \parallel [e_s], add(\sigma_3, x, 5))$, as shown in Figure 4.5. We start with Rule E.14, which adds the tagged triple

$$(r_s, ([e_s],\ add(\sigma_3, x, 5),\ [v_s]))$$

and directs the tracking to the run-triple $([e_s, 1, e_t],\ add(\sigma_3, x, 5),\ v_t)$, again for subvalue path $[1]$. We continue with Rule E.11, which adds

$$(r_s, ([e_s, 1, e_t],\ add(\sigma_3, x, 5),\ [v_t]))$$

and directs us to $([e_s, 1, e_t, e_g],\ add(\sigma_3, x, 5),\ \langle k\colon 25, l\colon 1, m\colon 11\rangle)$, for the subvalue path $[\langle k\colon 25, l\colon 1, m\colon 11\rangle, l, 1]$. As g is bound to an external service, we yet again use Rule E.13E, which stops the tracking process after adding

$$(r_s, ([e_s, 1, e_t, e_g],\ add(\sigma_3, x, 5),\ [\langle k\colon 25, l\colon 1, m\colon 11\rangle, l, 1]))\,.$$

Again, we have reached the run-triple from *where* φ_2 originates, in $run(r_s)$. The collected set of tagged triples is thus

$$\begin{aligned}
\{\, & (r, ([e],\ \sigma,\ [out, \langle b\colon 4, c\colon 1\rangle, c, 1])),\\
& (r, ([e, 2, e_1],\ add(\sigma, x, in_3),\ [v_3, c, 1])),\\
& (r, ([e, 2, e_1, c, e_3],\ add(\sigma, x, in_3),\ [w_3])),\\
& (r_s, ([e_s],\ add(\sigma_3, x, 5),\ [v_s])),\\
& (r_s, ([e_s, 1, e_t],\ add(\sigma_3, x, 5),\ [v_t])),\\
& (r_s, ([e_s, 1, e_t, e_g],\ add(\sigma_3, x, 5),\ [\langle k\colon 25, l\colon 1, m\colon 11\rangle, l, 1]))\,\}
\end{aligned}$$

Note that now we can again apply Rule E.13G instead of Rule E.13E to reach the free variable x of e_s. That would allow us to return to Rule E.13S, to continue the tracking process into $Prov^*([5], r \parallel [e, 2, e_1, c, e_3, 1, e_4], add(\sigma, x, in_3))$, back in the run of the parent dataflow.

On a final note, it would be interesting to develop a GUI, with a graphical representation of a run, that would facilitate exploring the run through the provenance inference rules.

5

Open Provenance Model

5.1 Overview of OPM

In this section we present a representative example to illustrate key components of the Open Provenance Model. For a detailed presentation of OPM, we refer to the OPM reference specification [MCF$^+$11]. Consider the following scenario:

> Alice and her young son Bob ordered a latte and a fruit juice in a coffee shop. Bob, who is a young child, did not observe the activities involved in processing their order, and only focused on his own drink. Hence, Bob's version of events is that an order was submitted and resulted in his juice being delivered.
>
> Alice, who could observe the activities behind the counter, identified three different processes. The cashier took the order and associated payment. As soon as the order was taken, the cashier put an empty cup on a tray next to the coffee machine; once payment was taken, the cashier added a till receipt to the same tray. The coffee machine operator picked up the cup, and filled it with the requested coffee, as per receipt, and handed the tray over to Alice. A third person behind the counter served other drinks, on request from the cashier. Alice was unable to ascertain how information was communicated (e.g., the request was stated by cashier, or order read from receipt); what is definite from Alice's viewpoint is the juice was also delivered with the tray.

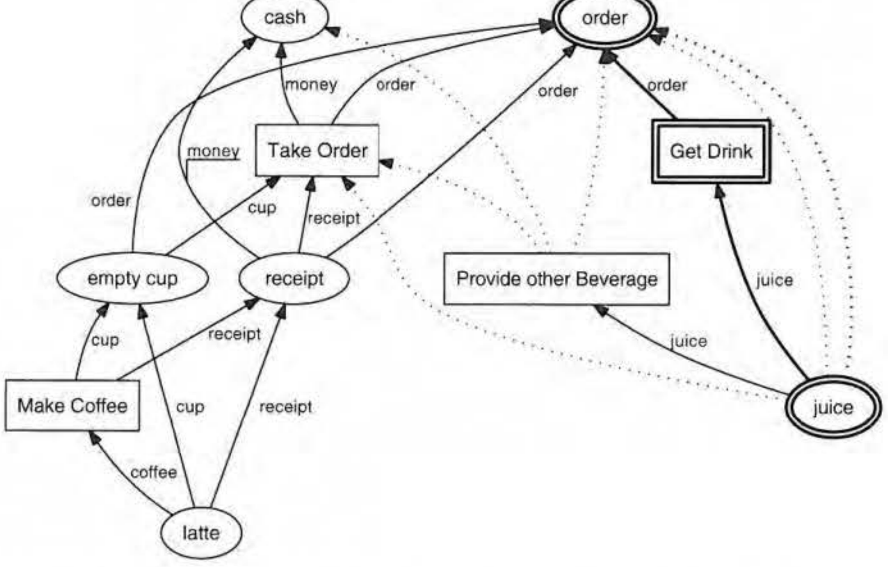


Figure 5.1: OPM graph for coffee shop order. The dotted edges are imprecise.

The OPM data model consists of a directed graph, whose nodes are artifacts and processes, and edges are dependencies between them. Artifacts in this scenario consist of an "order", some "cash", an "empty cup", a "receipt", a "juice" and a "latte". According to Bob, there is a single process: "Get Drink". Alice's version of events is more detailed and involves three processes: "Take Order", "Make Coffee", and "Provide other Beverages".

OPM edges are directional: an edge source represents an *effect* and an edge destination a *cause*. There exist four types of edges according to the types of effect and cause. A *used*-edge is between a process and an artifact; a *generated-by* edge is between an artifact and a process; a *derived-from* edge is between two artifacts; and an *informed-by* edge is between two processes.

Moreover, the first three types of edges are further categorised into *precise* and *imprecise* versions. Precise edges are labeled with *roles*, which indicate the role in which artifacts are used and generated; roles are comparable to parameter positions in a procedure. Imprecise edges are used when precise information about what happened is not important or not available.

Nodes are listed in Table 5.1, and edges in Table 5.2. Nodes and edges are displayed graphically in Figure 5.1. The "Take Order" process used two artifacts, "order" and "cash", and generated the "empty cup". The latter was used by the process "Make Coffee", to generate a "latte".

Table 5.1: OPM processes and artifacts for coffee shop order

node type	node	label
process	p_1	Take Order
process	p_2	Make Coffee
process	p_3	Provide other Beverage
process	p_4	Get Drink
artifact	a_1	order
artifact	a_2	cash
artifact	a_3	empty cup
artifact	a_4	receipt
artifact	a_5	latte
artifact	a_6	juice

We note that the empty cup was put on the tray, before payment was taken. So there is no edge from the "empty cup" to "cash". On the other hand, the receipt was put on the tray after cash was received, which explains the presence of an edge from "receipt" to "cash".

Furthermore, some OPM edges can be decorated with time information (not represented explicitly in the figure nor table). Specifically, the time associated with a precise used-edge denotes the time at which an artifact was used by a process; likewise, the time associated with a precise generated-by edge denotes the time at which an artifact was generated by a process. Moreover, processes may be given a beginning and an ending time.

OPM specifies some constraints between such time information and the graph structure. For instance, let t be the time associated with (p_1, money, a_2). Time t represents the time at which p_1 ("Take Order") used artifact a_2 ("cash"), with role "money". Time t is required to precede the ending of p_1, and to follow the beginning of p_1. Note also that p_1 may well be finished before artifact a_5 ("latte") was actually produced. This paper formalises all constraints identified by OPM.

We note that Bob's version of events is represented in the same graph as Alice's version. In the graphical representation of Figure 5.1, the nodes in Bob's version have a double border, and the edges are bold. Artifacts "order" and "juice" belong to both versions. To support multiple descriptions of an execution, OPM introduces a notion of account. An account is a subgraph, which is also an OPM graph.

Table 5.2: OPM edges for coffee shop order

edge type	source /effect	destination /cause	asserted edges for Figure 5.1
precise generated-by	artifact	process	(a_3, cup, p_1), $(a_4, \text{receipt}, p_1)$, $(a_5, \text{coffee}, p_2)$,$(a_6, \text{juice}, p_3)$ (a_6, juice, p_4)
precise used	process	artifact	(p_1, money, a_2), (p_1, order, a_1), $(p_2, \text{receipt}, a_4)$, (p_2, cup, a_3), (p_4, order, a_1)
precise derived-from	artifact	artifact	(a_3, order, a_1), (a_4, order, a_1), (a_4, money, a_2), (a_5, cup, a_3), $(a_5, \text{receipt}, a_4)$
generated-by	artifact	process	(a_6, p_1)
used	process	artifact	(p_3, a_1), (p_3, a_2)
derived-from	artifact	artifact	(a_6, a_1)
informed-by	process	process	(p_3, p_1)

5.2 OPM graphs and their temporal semantics

The OPM reference specification [MCF$^+$11] defines the proposed data model only informally. The purpose of this section is to provide a temporal semantics to OPM graphs, the data structure introduced in the reference specification.

Section A.4 contains a set of notes covering technical details of the OPM reference specification, and their relationship with the temporal semantics we propose. When relevant, we refer to these notes.

5.2.1 OPM graphs

We begin by formally defining OPM graphs. Our definition is slightly more detailed in distinguishing between precise and imprecise edges.

In our formalisation, OPM graphs consist of nodes and edges. Nodes can be of two types: artifacts and processes (Note A.4.1). There are four types of edges: generated-by, used, derived-from, and informed-by, depending on the type of their source and destination (Note A.4.2).

The edges are further categorised into *precise* and *imprecise* edges. Precise edges are syntactically marked by the presence of roles to characterise the nature of the relationship between the source and destination of the edge. Intuitively, OPM roles are to used-edges, what parameter positions are to procedures in programming languages; likewise, roles in a generated-by edge identify the nature of an output generated by a process; finally, roles in a derived-from edge characterise the precise usage of an artifact by a process. By contrast, imprecise edges do not have roles; they represent incomplete information (Note A.4.3).

This work provides a semantic interpretation of precise and imprecise edges.

Definition 5.1 (OPM graph). An OPM graph is a structure

$$(\mathit{Art}, \mathit{Proc}, \mathit{Roles}, \mathit{GeneratedBy}!, \mathit{Used}!, \mathit{DerivedFrom}!, \\ \mathit{GeneratedBy}, \mathit{Used}, \mathit{DerivedFrom}, \mathit{InformedBy})$$

where

- *Art* and *Proc* are two disjoint finite sets of elements called *artifacts* and *processes*, respectively;
- *Roles* is a finite set of elements called *roles*;
- $\mathit{GeneratedBy}! \subseteq \mathit{Art} \times \mathit{Roles} \times \mathit{Proc}$;
- $\mathit{Used}! \subseteq \mathit{Proc} \times \mathit{Roles} \times \mathit{Art}$;
- $\mathit{DerivedFrom}! \subseteq \mathit{Art} \times \mathit{Roles} \times \mathit{Art}$;
- $\mathit{GeneratedBy} \subseteq \mathit{Art} \times \mathit{Proc}$;
- $\mathit{Used} \subseteq \mathit{Proc} \times \mathit{Art}$;
- $\mathit{DerivedFrom} \subseteq \mathit{Art} \times \mathit{Art}$;
- $\mathit{InformedBy} \subseteq \mathit{Proc} \times \mathit{Proc}$.

Artifacts and processes are collectively referred to as the *nodes* of an OPM graph. The elements of $\mathit{GeneratedBy}! \cup \mathit{Used}! \cup \mathit{DerivedFrom}!$ are called *precise edges*, and the elements of $\mathit{GeneratedBy} \cup \mathit{Used} \cup \mathit{DerivedFrom} \cup \mathit{InformedBy}$ are called *imprecise edges*; all together they are called edges (Note A.4.4). Precise edges are of the form (x, r, y) and are additionally denoted as $x \xrightarrow{r} y$, or, when it is not important to know r, as $x \xrightarrow{!} y$. Imprecise edges (x, y) are denoted simply as $x \to y$. When the distinction between precise and imprecise derived-from edges is of no consequence, we use the following set to refer to all derived-from edges of an OPM graph:

$$\mathit{DerivedEdges} = \mathit{DerivedFrom} \cup \{(A, B) \mid (A, r, B) \in \mathit{DerivedFrom}!\} .$$

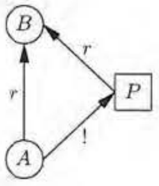


Figure 5.2: A use–generate-derive triangle (A, B, P, r).

This definition of graph allows for multiple precise used-edges between a same process-artifact pair with multiple roles. They would indicate that during its lifetime a process used a same artifact several times, with different roles.

The OPM reference specification defines a notion of legal graph as a directed graph without cycles in the derived-from edges, in which each artifact is generated by at most one process. For now, we relax the constraint on the derived-from edges, which we revisit in Section 5.4.2, and we refine this notion of legality in the context of our formalisation (Note A.4.5).

Definition 5.2 (Legal OPM graph). An OPM graph is called *legal* if

- for each artifact A there is at most one process P with a precise generated-by edge $A \xrightarrow{!} P$; and
- for each precise derived-from edge $A \xrightarrow{r} B$ there is a process P with precise edges $A \xrightarrow{!} P$ and $P \xrightarrow{r} B$, for the same role r.

A configuration (A, B, P, r) as above, with edges $A \xrightarrow{r} B$, $A \xrightarrow{!} P$, and $P \xrightarrow{r} B$, is called a *use–generate–derive triangle*, or simply *triangle* for short (see Figure 5.2). To denote that a use–generate–derive triangle (A, B, P, r) occurs in some given OPM graph G, we use the notation $G \bigtriangleup (A, B, P, r)$.

A use–generate–derive triangle offers an insight into the inner workings of a process P, since not only does it state that B was used by P in role r and A generated according to a role, but also does it state that B had a direct influence on A, because it was used in this precise role r.* A typical usage of a use–generate–derive triangle is for a division process, illustrated in the following example.

*The usage role in the use–generate–derive triangle is crucial. We could imagine an extension of Figure 5.2, in which P uses B in a second role, say s. The triangle of Figure 5.2 identifies the precise usage of B that affected the output A, here r, whereas, an alternate use of B, with role s, could have not impacted A (for instance, because it took place after A was created).

Example 5.3. Let / be a division process, 8 and 4 be its inputs (in respective capacity of dividend and divisor), and the quotient 2 be its output. So, edges are as follows:

edge type	source	destination	
precise generated-by	artifact	process	$(2, quotient, /)$
precise used	process	artifact	$(/, dividend, 8)$, $(/, divisor, 4)$
precise derived-from	artifact	artifact	$(2, dividend, 8)$, $(2, divisor, 4)$

They form two triangles: $(2, 8, /, dividend)$ and $(2, 4, /, divisor)$. □

In the following, unless otherwise explicitly stated, we only consider legal OPM graphs. Whenever we refer to a single OPM graph G, we use the names defined in this section to refer to the different constituents of the OPM graph. If we handle more than one OPM graph, for instance graphs G and H, we use superscripts G and H to distinguish their respective constituents. We extend this convention to other concepts related to OPM graphs.

5.2.2 Temporal models for OPM graphs

The OPM reference specification [MCF$^+$11] allows OPM graphs to be decorated with time information for specific time-points, which are meaningful in the context of a computation. Four of these are identified: the beginning of a process, the ending of a process, the instant a process uses an artifact, and the moment a process creates an artifact. Such time information is routinely captured by computer systems. For instance, creation time is readily available from file systems in typical operating systems. HTTP servers and databases logs would usually include the time at which a document or table is read or queried, respectively. Likewise, the beginning and ending time of processes are frequently recorded by job submission systems. The OPM reference specification introduces some constraints between time-points, such as an artifact can only be used after it has been created. In this section, we revisit the notion of time in OPM, by means of a temporal interpretation of a graph, in terms of all the time constraints that it implies (Note A.4.6).

A *temporal interpretation* of a legal OPM graph is an assignment of the following time-points to processes, artifacts, and precise used-edges:[†] (Note A.4.7)

[†]One may wonder why precise generated-by edges do not get a time-point. But, as a matter of fact, they do. For each precise generated-by edge $A \xrightarrow{r} P$, we indeed have a time-point create(A). Since the OPM graph is legal, there can be at most one precise edge emanating from A, so we do not need to specify r and P.

- for each artifact A, its creation time, denoted by create(A);
- for each process P, its beginning and ending times, denoted by begin(P) and end(P) respectively;
- for each precise used-edge $P \xrightarrow{r} A$, the time when P "read" A in role r, denoted by use(P, r, A).

Formally, we fix some OPM graph G for the remainder of this section. We define the set of *temporal variables* of G, denoted by $\mathit{Vars}(G)$, as follows:

$$\mathit{Vars}(G) = \{\mathrm{create}(A) \mid A \in \mathit{Art}\} \cup \{\mathrm{begin}(P), \mathrm{end}(P) \mid P \in \mathit{Proc}\} \\ \cup \{\mathrm{use}(P, r, A) \mid (P, r, A) \in \mathit{Used!}\}.$$

We then define:

Definition 5.4. A *temporal interpretation* of G is a triple $(T, \leq, \tau)$, where

- T is a set, we call its elements *time-points*;
- $\leq$ is a partial order on T;
- τ is a mapping from $\mathit{Vars}(G)$ to T.

When no confusion can arise, we omit T and $\leq$ from the notation and simply denote a temporal interpretation by τ.

Not every temporal interpretation makes sense as a temporal model of G. Indeed, to reflect the dependencies specified in G, the interpretation should satisfy various constraints reflecting these dependencies.

In order to define these constraints formally, we define an *inequality* over G as an expression of the form $u \preceq v$, with $u, v \in \mathit{Vars}(G)$. By a *trivial* inequality we mean an inequality of the form $u \preceq u$. We are now ready to define the set of constraints expressed by a legal OPM graph.

Definition 5.5. The *temporal theory* of G, denoted by Th(G), is the set consisting of all the inequalities stated in the following *axioms*:

AX 1: for each process P, the inequality begin$(P) \preceq$ end(P);

AX 2: for each precise generated-by edge $A \xrightarrow{!} P$ in G, the inequalities begin$(P) \preceq$ create$(A) \preceq$ end(P);

AX 3: for each precise used-edge $P \xrightarrow{r} A$ in G, the three inequalities begin$(P) \preceq$ use(P, r, A), use$(P, r, A) \preceq$ end(P), and create$(A) \preceq$ use(P, r, A);

AX 4: for each imprecise derived-from edge $A \rightarrow B$ in G, the inequality $\text{create}(B) \preceq \text{create}(A)$;

AX 5: for each imprecise generated-by edge $A \rightarrow P$ in G, the inequality $\text{begin}(P) \preceq \text{create}(A)$;

AX 6: for each imprecise used-edge $P \rightarrow A$ in G, the inequality $\text{create}(A) \preceq \text{end}(P)$;

AX 7: for each informed-by edge $P \rightarrow Q$ in G, the inequality $\text{begin}(Q) \preceq \text{end}(P)$;

AX 8: for each $G \triangle (A, B, P, r)$, the inequality $\text{use}(P, r, B) \preceq \text{create}(A)$.

Axioms 1–7 are either obvious (e.g., Axiom 1) or are in line with the OPM reference specification (Note A.4.8). The eighth axiom, the "triangle axiom", corresponds to the intended usage of OPM by which a precise derived-from edge in a generate–use–derive triangle (A, B, P, r) in G is not redundant, but expresses exactly that P needed to read B in role r before it could generate A (Note A.4.9).

Example 5.6. The temporal interpretation of precise edges can be illustrated by a service analogy. Let us consider a translation Web Service P. The service has to be running to receive a translation request A and for the translation result B to be returned; so A is received (used) and B is sent (created) after the beginning of P and before its end; furthermore, B is created after A is received. □

Example 5.7. Referring to Figure 5.1, the coffee machine operator begins the "Make Coffee" process by cleaning the steam pipe and emptying the coffee filter; once an "empty cup" and "receipt" are available, they are used (precise edge) to generate a "latte" (precise edge). In the same figure, it is unspecified when the "order" is taken, with respect to the beginning of the "Provide Beverages" process; hence, an imprecise used-edge appears in the figure. □

We finally define the temporal models of G as follows. Naturally, a temporal interpretation τ is said to *satisfy* an inequality $u \preceq v$ if $\tau(u) \leq \tau(v)$.

Definition 5.8. A temporal interpretation τ of G is a *temporal model* of G, denoted by $\tau \models \text{Th}(G)$, if it satisfies all inequalities from $\text{Th}(G)$.

Example 5.9. Consider the small OPM graph G shown in Figure 5.2. Let us use natural numbers with their natural ordering as time-points. Then the two interpretations τ_1 and τ_2, presented in Table 5.3, are temporal models of G.

Table 5.3: Two temporal models for the graph shown in Figure 5.2.

τ_1	variable	value
	create(B)	1
	begin(P)	2
	use(P, r, B)	3
	create(A)	4
	end(P)	5

τ_2	variable	value
	create(B)	1
	begin(P)	1
	use(P, r, B)	1
	create(A)	1
	end(P)	1

Temporal model τ_2, which maps all temporal variables to the same time-point, might be generated by a very coarse clock.

Many temporal interpretations of G, however, are not temporal models of G. If, for example, we would modify τ_1 to τ_1' by setting $\tau_1'(\text{end}(P)) = 0$, then Axiom 1 would be violated. Likewise, if we would modify τ_2 to τ_2' by setting $\tau_2'(\text{use}(P, r, B)) = 0$, then Axiom 3 would be violated. Also, if we would modify τ_1 to τ_1'' by setting $\tau_1''(\text{create}(A)) = 0$, then we would violate Axioms 2 and 8. □

Example 5.10. For another example, consider an OPM graph with two artifacts A and B and nothing else (no edges either). Then any possible temporal interpretation qualifies as a model. In particular, in some models τ we have $\tau(\text{create}(A)) < \tau(\text{create}(B))$; in other models we have $\tau(\text{create}(B)) < \tau(\text{create}(A))$; and still in others we have $\tau(\text{create}(A)) = \tau(\text{create}(B))$. This is because the OPM graph does not impose any constraints by the absence of any edges. □

Whilst the temporal semantics is a novel contribution, the OPM reference specification [MCF$^+$11] defines time placeholders in some constructs, and allows them to be filled with time information. These time-decorated constructs correspond to the time variables introduced in this work. The OPM reference specification does not mandate all time placeholders to be filled. Thus, from a temporal semantics viewpoint, for every decorated construct, the time information found in the placeholder fixes τ for the corresponding variable. If all placeholders are filled, then a single temporal interpretation exists. In general, for every filled placeholder, the number of possible interpretations is reduced.

Now that we have formally defined a temporal model for OPM graphs, we can investigate, in the next section, how we can conduct inference in OPM graphs. Whether there are other, non-temporal, ways to provide a formal semantics for OPM graphs is an interesting direction for further research. (We briefly discuss other efforts in Chapter 1).

5.3 Inference in OPM graphs

The axioms of Definition 5.5 allow us to obtain a number of inequalities over an OPM graph's variables. These inequalities logically imply further inequalities. For a trivial example, in an OPM graph with derived-from edges $A \to B \to C$, Axiom 4 gives the inequalities $\text{create}(C) \preceq \text{create}(B)$ and $\text{create}(B) \preceq \text{create}(A)$, which logically imply the further inequality $\text{create}(C) \preceq \text{create}(A)$.

Formally, we define:

Definition 5.11. Let G be a legal OPM graph and let $u, v \in \mathit{Vars}(G)$. The inequality $u \preceq v$ is a *logical consequence* of G, denoted by $\text{Th}(G) \models u \preceq v$, if $u \preceq v$ is satisfied in every temporal model of G.

A general example of logical consequence is provided by the following lemma and proof.

Lemma 5.12. *Let G be a legal OPM graph with artifacts A and B and a precise edge $A \xrightarrow{r} B$ for some role r. Then* $\text{Th}(G) \models \text{create}(B) \preceq \text{create}(A)$.

Before proving this lemma we note that Axiom 4 is almost exactly the same, except that it is stated for an imprecise derived-from edge. Thus, the present lemma shows that the same constraint holds for precise derived-from edges. This constraint did not need to be explicitly given as an axiom because it already logically follows from the given axioms.

Proof. Since G is legal, there exists a process P in G with edges $P \xrightarrow{r} B$ and $A \xrightarrow{!} P$. Let τ be a temporal model of G. By Axiom 3 we have $\tau(\text{create}(B)) \leq \tau(\text{use}(P, r, B))$. By Axiom 2 we have $\tau(\text{use}(P, r, B)) \leq \tau(\text{create}(A))$. We conclude $\tau(\text{create}(B)) \leq \tau(\text{create}(A))$ as desired. □

One may indeed wonder exactly which inequalities logically follow from a given OPM graph. It is well known that an inequality $u \preceq w$ can be inferred from $\text{Th}(G)$ by using repeated applications of the rule of transitivity: "from $u \preceq v$ and $v \preceq w$ we infer $u \preceq w$".[‡] However, this way it is hard to relate

[‡] For a set of inequalities Σ and an inequality φ, φ is a logical consequence of Σ if and only if φ can be inferred from Σ by using transitivity. Ullman [Ull90] presents a self-contained proof for a more general case.

the newly inferred inequalities to nodes and edges in the graph. Fortunately, we show in Section 5.3.2 that it is possible to perform temporal inference in a purely graphical manner. We prove in Theorem 5.14 that every possible logical consequence can be directly inferred from the OPM graph by looking for a fixed set of patterns in the graph.

5.3.1 Edge-inference rules

The cornerstone of our graph-based inference of inequalities is provided by four inference rules that infer new edges in an OPM graph. These four rules are already part of the OPM reference specification [MCF+11], except that here we extend them to better take precise edges into account. Inferred edges prove to play an important role in graphical patterns that we introduce to infer inequalities. Moreover, we establish that inference of edges is the only action we need to perform to infer inequalities that do not involve use-variables. (For inequalities involving use-variables, patterns more complicated than a single edge have to be matched in the graph.) We thus provide a justification for the edge inferences introduced in the OPM reference specification.

We first introduce the inference of edges at an intuitive level. Then we define it formally in Definition 5.13. According to the OPM reference specification, the edges present in an OPM graph G denote dependencies. From the given dependencies in G, we can infer derived dependencies. A very intuitive type of inference is to follow chains of derived-from edges. In this section, we define edge-inference rules based on this intuition.

Suppose there is a chain of derived-from edges in G (which can be either precise or imprecise) that starts in an artifact A and ends in an artifact C. We denote this by $A \dashrightarrow C$. Formally, relation $\dashrightarrow$ between two artifacts is nothing else than the transitive closure of *DerivedEdges*. Since A has been indirectly derived from C, we can think of $A \dashrightarrow C$ as an inferred edge, as illustrated in Figure 5.3(a).

Next we show how to infer generated-by edges. Suppose we have artifacts A and B with $A \dashrightarrow B$ and, in addition, a generated-by edge from B to a process P in G, either a precise edge $B \xrightarrow{!} P$ or an imprecise edge $B \rightarrow P$. Then A has been indirectly generated by P and we can infer an edge $A \dashrightarrow P$, as illustrated in Figure 5.3(b).

We can infer used-edges as well. Suppose we have artifacts A and B with $A \dashrightarrow B$. In addition, there is a used-edge from a process P to A in G, either precise $P \xrightarrow{!} A$ or imprecise $P \rightarrow A$. Then P has indirectly used B and we can infer an edge $P \dashrightarrow B$, as illustrated in Figure 5.3(c). Moreover, we

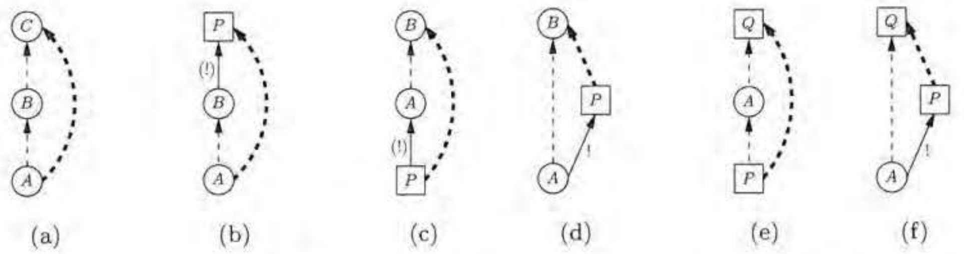


Figure 5.3: Inference of (a) derived-from, (b) generated-by, (c)–(d) used and (e)–(f) informed-by edges. The bold edges are newly inferred. The edges labeled by "(!)" may be either precise or imprecise.

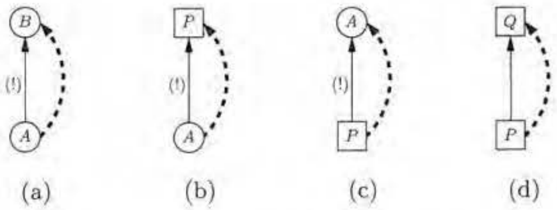


Figure 5.4: Trivial inference of (a) derived-from, (b) generated-by, (c) used and (d) informed-by edges.

can also infer a used-edge in the following situation. Suppose we again have $A \dashrightarrow B$, but now in combination with a precise edge $A \xrightarrow{!} P$ in G. Since A was precisely generated by P, but A has also been indirectly derived from B, we can conclude that P has indirectly used B. Again, we can infer $P \dashrightarrow B$, which we show in Figure 5.3(d).

Finally, to infer informed-by edges, we can reason as follows. Suppose, for some processes P and Q and an artifact A, we have edges $P \dashrightarrow A$ and $A \dashrightarrow Q$, which are already present in G (either precise or imprecise) or have been previously inferred. Then A represents information that flowed from Q to P and we can infer an edge $P \dashrightarrow Q$, as illustrated in Figure 5.3(e). Moreover, an informed-by edge can also be inferred in the following case. Suppose we again have $A \dashrightarrow Q$, but now in combination with a precise edge $A \xrightarrow{!} P$ in G. Since A was directly generated by P, but A was also indirectly generated by Q, we can conclude that P was somehow influenced by Q. Again, we can infer $P \dashrightarrow Q$, which we show in Figure 5.3(f).

There are also trivial inferences for all types of edges, to the effect that an edge that is already present in the graph can always be inferred, as illustrated in Figure 5.4.

$$\frac{A \to B \text{ in } G \text{ or } A \stackrel{!}{\to} B \text{ in } G}{G \vdash A \dashrightarrow B}\ \text{TRIVIAL DERIVED-FROM}$$

$$\frac{A \to P \text{ in } G \text{ or } A \stackrel{!}{\to} P \text{ in } G}{G \vdash A \dashrightarrow P}\ \text{TRIVIAL GENERATED-BY}$$

$$\frac{P \to A \text{ in } G \text{ or } P \stackrel{!}{\to} A \text{ in } G}{G \vdash P \dashrightarrow A}\ \text{TRIVIAL USED}$$

$$\frac{P \to Q \text{ in } G}{G \vdash P \dashrightarrow Q}\ \text{TRIVIAL INFORMED-BY}$$

Figure 5.5: Trivial edge-inference rules.

The above discussion is formalised in the following definition.

Definition 5.13 (Edge-inference rules). Let G be a legal OPM graph and let X and Y be two nodes in G. In the following, we define when $X \dashrightarrow Y$ can be *inferred* from G, denoted by $G \vdash X \dashrightarrow Y$. Specifically, let A, B and C be artifacts in G, and let P and Q be processes in G.

We begin by stating four trivial inference rules which mean that if an edge already belongs to G, then that edge can be inferred from G. These rules are presented in Figure 5.5. Next we define four further inference rules, in cases where at least one of the present edges was previously inferred. These rules are presented in Figure 5.6.

Note that, as a direct consequence of the above definition, we have the following properties:

- $G \vdash A \dashrightarrow B$ iff (A, B) belongs to the transitive closure of *DerivedEdges*;
- if $G \vdash A \dashrightarrow B$ and $G \vdash B \dashrightarrow P$ then $G \vdash A \dashrightarrow P$;
- if $G \vdash P \dashrightarrow A$ and $G \vdash A \dashrightarrow B$ then $G \vdash P \dashrightarrow B$.

Edge-inference rules introduced in this section allow us to derive new edges from a graph G, noted as $G \vdash X \dashrightarrow Y$, with X and Y two nodes of G. Inferred edges do not belong to the sets of edges identified in Definition 5.1, implying that these edges $X \dashrightarrow Y$ are inferred "outside" G. Thus, the temporal theory

$$\frac{G \vdash A \dashrightarrow B \qquad G \vdash B \dashrightarrow C}{G \vdash A \dashrightarrow C}\ \textsc{derived-from}$$

$$\frac{G \vdash A \dashrightarrow B \qquad B \rightarrow P \text{ in } G \text{ or } B \overset{!}{\rightarrow} P \text{ in } G}{G \vdash A \dashrightarrow P}\ \textsc{generated-by}$$

$$\frac{G \vdash A \dashrightarrow B \qquad P \rightarrow A \text{ in } G \text{ or } P \overset{!}{\rightarrow} A \text{ in } G \text{ or } A \overset{!}{\rightarrow} P \text{ in } G}{G \vdash P \dashrightarrow B}\ \textsc{used}$$

$$\frac{G \vdash A \dashrightarrow Q \qquad G \vdash P \dashrightarrow A \text{ or } A \overset{!}{\rightarrow} P \text{ in } G}{G \vdash P \dashrightarrow Q}\ \textsc{informed-by}$$

Figure 5.6: Edge-inference rules.

of Definition 5.5 does not directly associate a temporal meaning to these edges. In the next section, we observe that inferred edges have a similar temporal semantics as imprecise edges.

5.3.2 Characterisation of temporal inference

Let us reconsider the axioms of Definition 5.5 that define the temporal semantics of an OPM graph. We see that each axiom is a rule that relates a pattern in the graph to one or more inequalities. For example, Axiom 2 relates the pattern consisting simply of a single edge $A \overset{!}{\rightarrow} P$, to the inequalities $\text{begin}(P) \preceq \text{create}(A)$ and $\text{create}(A) \preceq \text{end}(P)$. Axiom 1 even relates the pattern consisting simply of a process node P to the inequality $\text{begin}(P) \preceq \text{end}(P)$. Axiom 8 has a more complicated pattern in the form of a use–generate–derive triangle.

In a similar way, we now introduce ten more such rules. Rules 1–9A–9B are shown in Figure 5.7. (The figure also includes some axioms, but we explain this after the statement of Theorem 5.14.) An important difference with the axioms, however, is that every dashed edge in a pattern now stands not just for an edge that is present in the graph, but for an edge that can be inferred by the edge-inference rules.

The following theorem states that these rules are *sound* and *complete* in the following sense. The rules are sound in that they represent valid inferences: the inequalities they infer are indeed logical consequences of the axioms in the sense of Definition 5.11. Moreover, the rules are complete in that any inequality that is a logical consequence of the axioms, and that is not already part of the axioms, can be inferred by one of the ten rules.

Theorem 5.14. *Let G be a legal OPM graph and let φ be a nontrivial inequality over the temporal variables of G. Then* $\mathrm{Th}(G) \models \varphi$ *if and only if either (0) φ already belongs to* $\mathrm{Th}(G)$, *or φ matches one of the following inequalities:*

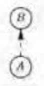
- *Cases* not *involving use-variables:*
  1. $\mathrm{create}(B) \preceq \mathrm{create}(A)$ *with* $G \vdash A \dashrightarrow B$;
  2. $\mathrm{begin}(P) \preceq \mathrm{create}(A)$ *with* $G \vdash A \dashrightarrow P$;
  3. $\mathrm{create}(A) \preceq \mathrm{end}(P)$ *with* $G \vdash P \dashrightarrow A$;
  4. $\mathrm{begin}(Q) \preceq \mathrm{end}(P)$ *with* $G \vdash P \dashrightarrow Q$;
- *Cases involving use-variables:*
  5. $\mathrm{create}(B) \preceq \mathrm{use}(P, r, A)$ *with* $P \xrightarrow{r} A$ *in G and* $G \vdash A \dashrightarrow B$;
  6. $\mathrm{begin}(Q) \preceq \mathrm{use}(P, r, A)$ *with* $P \xrightarrow{r} A$ *in G and* $G \vdash A \dashrightarrow Q$;
  7. $\mathrm{use}(P, r, C) \preceq \mathrm{create}(A)$ *with* $G \bigtriangleup (B, C, P, r)$ *for some B, and* $G \vdash A \dashrightarrow B$;
  8. $\mathrm{use}(P, r, B) \preceq \mathrm{end}(Q)$ *with* $G \bigtriangleup (A, B, P, r)$ *for some A, and* $G \vdash Q \dashrightarrow A$;
  9. $\mathrm{use}(P, r, B) \preceq \mathrm{use}(Q, s, A)$ *with* $G \bigtriangleup (C, B, P, r)$ *for some C, with* $Q \xrightarrow{s} A$ *in G, and either (a)* $A = C$ *or (b)* $G \vdash A \dashrightarrow C$.

Note that in the above, A, B and C, or P and Q, need not be distinct.

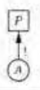
Since Rules 1–4 subsume Axioms 4–7, Figure 5.7, which includes the remaining axioms, provides a complete picture of the possible logical consequences of an OPM graph in the sense of Definition 5.11. That definition was purely semantic and does not give any concrete algorithm for checking logical consequence. The figure now gives us direct shortcuts from patterns in an OPM graph to its logical consequences.

The inference rules of Figure 5.7 are an entirely novel characterisation of the temporal inferences of OPM since they are sound and complete, in the sense defined in this section. To check that an inequality $u \preceq v$ is logical consequence of a graph, it suffices to select the corresponding pattern in Figure 5.7, and verify that it is satisfied by the graph (extended with the proper inferred edges). Vice versa, if an inequality $u \preceq v$ is logical consequence of $\mathrm{Th}(G)$, then the corresponding pattern is known to exist in G.

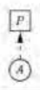
(AX 1) $\text{begin}(P) \preceq \text{end}(P)$



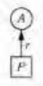
(1) $\text{create}(B) \preceq \text{create}(A)$



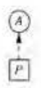
(AX 2) $\text{begin}(P) \preceq \text{create}(A) \preceq \text{end}(P)$



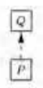
(2) $\text{begin}(P) \preceq \text{create}(A)$



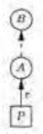
(AX 3) $\text{begin}(P) \preceq \text{use}(P, r, A) \preceq \text{end}(P) \quad \text{create}(A) \preceq \text{use}(P, r, A)$



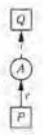
(3) $\text{create}(A) \preceq \text{end}(P)$



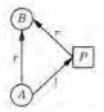
(4) $\text{begin}(Q) \preceq \text{end}(P)$



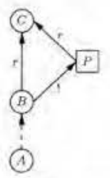
(5) $\text{create}(B) \preceq \text{use}(P, r, A)$



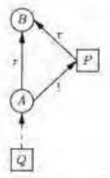
(6) $\text{begin}(Q) \preceq \text{use}(P, r, A)$



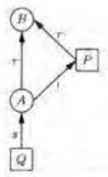
(AX 8) $\text{use}(P, r, B) \preceq \text{create}(A)$



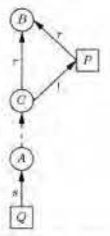
(7) $\text{use}(P, r, C) \preceq \text{create}(A)$



(8) $\text{use}(P, r, B) \preceq \text{end}(Q)$



(9A) $\text{use}(P, r, B) \preceq \text{use}(Q, s, A)$



(9B) $\text{use}(P, r, B) \preceq \text{use}(Q, s, A)$

Figure 5.7: Characterisation of temporal inference.

We anticipate that developers can leverage Theorem 5.14 to design reasoners for OPM. So far, reasoners have typically relied on Semantic Web technologies, such as OWL and SRWL, to compute transitive closures of OPM properties [MDF$^+$10, MF08]. What this theorem shows is that there are logical consequences involving use-variables that cannot be directly represented by edges in OPM graphs.

5.3.3 Proof of Theorem 5.14

In this section we present the proof of Theorem 5.14. First, we tackle the soundness property; then, we address the completeness proposition.

Proof of soundness

Let G be a legal OPM graph and let φ be a nontrivial inequality over the temporal variables of G that satisfies the conditions from Theorem 5.14. We have to show that $\mathrm{Th}(G) \models \varphi$. Thereto, let τ be a temporal model of G, i.e., $\tau \models \mathrm{Th}(G)$. We have to show that τ satisfies φ. We inspect the ten possibilities for φ:

(0) if $\varphi \in \mathrm{Th}(G)$, then τ satisfies φ since $\tau \models \mathrm{Th}(G)$.

(1) φ is $\mathrm{create}(B) \preceq \mathrm{create}(A)$ with $G \vdash A \dashrightarrow B$.

As a consequence of Definition 5.13, $G \vdash A \dashrightarrow B$ holds if (A, B) belongs to the transitive closure of *DerivedEdges*. Therefore, there is a path $A_1, A_2, \ldots, A_n$ of derived-from edges from A to B, for some $n \geq 2$ with $A_1 = A$ and $A_n = B$, and with $(A_i, A_{i+1}) \in \mathit{DerivedEdges}$, for $i \in \{1, \ldots, n-1\}$. Since every (A_i, A_{i+1}) is an edge in G, we know that $\mathrm{create}(A_{i+1}) \preceq \mathrm{create}(A_i)$ belongs to $\mathrm{Th}(G)$ (Axiom 4 and Lemma 5.12) and is thus satisfied by τ, i.e., $\tau(\mathrm{create}(A_{i+1})) \leq \tau(\mathrm{create}(A_i))$. Hence we also have $\tau(\mathrm{create}(A_n)) \leq \tau(\mathrm{create}(A_1))$, because $\leq$ is a partial order for τ. Thus τ satisfies $\mathrm{create}(B) \preceq \mathrm{create}(A)$.

(2) φ is $\mathrm{begin}(P) \preceq \mathrm{create}(A)$ with $G \vdash A \dashrightarrow P$.

By Definition 5.13, $G \vdash A \dashrightarrow P$ if either

a) there is already an edge $A \rightarrow P$ or $A \overset{!}{\rightarrow} P$ in G; or

b) there is an artifact B such that $G \vdash A \dashrightarrow B$ and there is an edge $B \rightarrow P$ or $B \overset{!}{\rightarrow} P$ in G.

2a) For an edge $A \to P$ ($A \xrightarrow{!} P$) in G, we know by Axiom 5 (Axiom 2), that $\varphi \in \mathrm{Th}(G)$ and thus τ satisfies φ.

2b) We already know from case 1 that τ satisfies $\mathrm{create}(B) \preceq \mathrm{create}(A)$ for $G \vdash A \dashrightarrow B$, i.e., we have $\tau(\mathrm{create}(B)) \leq \tau(\mathrm{create}(A))$. For an edge $B \to P$ ($B \xrightarrow{!} P$) in G, we know by Axiom 5 (Axiom 2), that $\mathrm{begin}(P) \preceq \mathrm{create}(B)$ belongs to $\mathrm{Th}(G)$. Therefore τ satisfies $\mathrm{begin}(P) \preceq \mathrm{create}(B)$, i.e., $\tau(\mathrm{begin}(P)) \leq \tau(\mathrm{create}(B))$. Hence $\tau(\mathrm{begin}(P)) \leq \tau(\mathrm{create}(A))$, since $\leq$ is a partial order for τ. We conclude that τ satisfies $\mathrm{begin}(P) \preceq \mathrm{create}(A)$.

(3) φ is $\mathrm{create}(A) \preceq \mathrm{end}(P)$ with $G \vdash P \dashrightarrow A$.

By Definition 5.13, $G \vdash P \dashrightarrow A$ if either

a) there is already an edge $P \to A$ or $P \xrightarrow{!} A$ in G; or

b) there is an artifact B such that $G \vdash B \dashrightarrow A$ and there is an edge $P \to B$ or $P \xrightarrow{!} B$ or $B \xrightarrow{!} P$ in G.

3a) For an edge $P \to A$ ($P \xrightarrow{!} A$) in G, we know by Axiom 6 (Axiom 3) that $\varphi \in \mathrm{Th}(G)$ and thus τ satisfies φ.

3b) We already know from case 1 that τ satisfies $\mathrm{create}(A) \preceq \mathrm{create}(B)$ for $G \vdash B \dashrightarrow A$. For an edge $P \to B$ ($P \xrightarrow{!} B$) in G, we know by Axiom 6 (Axiom 3) that $\mathrm{create}(B) \preceq \mathrm{end}(P)$ belongs to $\mathrm{Th}(G)$. For an edge $B \xrightarrow{!} P$ in G, we know by Axiom 2 that $\mathrm{create}(B) \preceq \mathrm{end}(P)$ belongs to $\mathrm{Th}(G)$. Therefore, in each case, τ satisfies both $\mathrm{create}(A) \preceq \mathrm{create}(B)$ and $\mathrm{create}(B) \preceq \mathrm{end}(P)$. Hence τ also satisfies $\mathrm{create}(A) \preceq \mathrm{end}(P)$.

(4) φ is $\mathrm{begin}(Q) \preceq \mathrm{end}(P)$ with $G \vdash P \dashrightarrow Q$.

By Definition 5.13, $G \vdash P \dashrightarrow Q$ if either

a) there is already an edge $P \to Q$ in G; or

b) there is an artifact A such that $G \vdash A \dashrightarrow Q$, and either $G \vdash P \dashrightarrow A$ or there is an edge $A \xrightarrow{!} P$ in G.

4a) For an edge $P \to Q$ in G, we know by Axiom 7 that $\varphi \in \mathrm{Th}(G)$ and thus τ satisfies φ.

4b) We already know from case 2 that τ satisfies $\mathrm{begin}(Q) \preceq \mathrm{create}(A)$ for $G \vdash A \dashrightarrow Q$. We also know from case (3) that τ satisfies $\mathrm{create}(A) \preceq \mathrm{end}(P)$ for $G \vdash P \dashrightarrow A$. For an edge $A \xrightarrow{!} P$ in G, we know by Axiom 2 that $\text{create}(A) \preceq \text{end}(P)$ belongs to $\text{Th}(G)$. Therefore, in each case, τ satisfies both $\text{begin}(Q) \preceq \text{create}(A)$ and $\text{create}(A) \preceq \text{end}(P)$. Thus τ also satisfies $\text{begin}(Q) \preceq \text{end}(P)$.

(5) φ is $\text{create}(B) \preceq \text{use}(P, r, A)$ with $P \xrightarrow{r} A$ in G and $G \vdash A \dashrightarrow B$.

We already know from case 1 that τ satisfies $\text{create}(B) \preceq \text{create}(A)$ for $G \vdash A \dashrightarrow B$. For edge $P \xrightarrow{r} A$ in G we know, by Axiom 3, that $\text{create}(A) \preceq \text{use}(P, r, A)$ belongs to $\text{Th}(G)$, and is thus satisfied by τ. Therefore, τ also satisfies $\text{create}(B) \preceq \text{use}(P, r, A)$.

(6) φ is $\text{begin}(Q) \preceq \text{use}(P, r, A)$ with $P \xrightarrow{r} A$ in G and $G \vdash A \dashrightarrow Q$.

We already know from case 2 that τ satisfies $\text{begin}(Q) \preceq \text{create}(A)$ for $G \vdash A \dashrightarrow Q$. For edge $P \xrightarrow{r} A$ in G, we know, by Axiom 3, that $\text{create}(A) \preceq \text{use}(P, r, A)$ belongs to $\text{Th}(G)$, and is thus satisfied by τ. Thus, τ also satisfies $\text{begin}(Q) \preceq \text{use}(P, r, A)$.

(7) φ is $\text{use}(P, r, C) \preceq \text{create}(A)$ with $G \triangle (B, C, P, r)$ and $G \vdash A \dashrightarrow B$.

We already know from case 1 that τ satisfies $\text{create}(B) \preceq \text{create}(A)$ for $G \vdash A \dashrightarrow B$. From $G \triangle (B, C, P, r)$ we know, by Axiom 8, that $\text{use}(P, r, C) \preceq \text{create}(B)$ belongs to $\text{Th}(G)$, and is thus satisfied by τ. Therefore, τ also satisfies $\text{use}(P, r, C) \preceq \text{create}(A)$.

(8) φ is $\text{use}(P, r, B) \preceq \text{end}(Q)$ with $G \triangle (A, B, P, r)$ and $G \vdash Q \dashrightarrow A$.

We already know from case 3 that τ satisfies $\text{create}(A) \preceq \text{end}(Q)$ for $G \vdash Q \dashrightarrow A$. From $G \triangle (A, B, P, r)$ we know, by Axiom 8, that $\text{use}(P, r, B) \preceq \text{create}(A)$ belongs to $\text{Th}(G)$, and is thus satisfied by τ. Hence, τ also satisfies $\text{use}(P, r, B) \preceq \text{end}(Q)$.

(9) φ is $\text{use}(P, r, B) \preceq \text{use}(Q, s, A)$ with $G \triangle (C, B, P, r)$ in G, $Q \xrightarrow{s} A$ in G, and either (a) $A = C$ or (b) $G \vdash A \dashrightarrow C$.

We already know from case 1 that τ satisfies $\text{create}(C) \preceq \text{create}(A)$ for $G \vdash A \dashrightarrow C$ (9b). If $A = C$ (9a) then, obviously, $\tau(A) = \tau(C)$, and τ still satisfies $\text{create}(C) \preceq \text{create}(A)$. For edge $Q \xrightarrow{s} A$ in G, we know, by Axiom 3, that $\text{create}(A) \preceq \text{use}(Q, s, A)$ belongs to $\text{Th}(G)$, and is thus satisfied by τ. From $G \triangle (C, B, P, r)$ we know, by Axiom 8S, that $\text{use}(P, r, B) \preceq \text{create}(C)$ belongs to $\text{Th}(G)$, hence is satisfied by τ. Therefore, we have $\text{use}(P, r, B) \preceq \text{create}(C) \preceq \text{create}(A) \preceq \text{use}(Q, s, A)$. We conclude that τ also satisfies $\text{use}(P, r, B) \preceq \text{use}(Q, s, A)$.

Proof of completeness

Let G be a legal OPM graph and let φ be a nontrivial inequality over the temporal variables of G such that $\mathrm{Th}(G) \models \varphi$. We must show that $\varphi \in \mathrm{Th}(G)$ or φ matches one of the cases 1–9 of Theorem 5.14.

It is well known [Ull90] that φ can be inferred from $\mathrm{Th}(G)$ by using repeated applications of the rule of transitivity: "from $u \preceq v$ and $v \preceq w$ infer $u \preceq w$." We proceed by induction on the number of applications of the transitivity rule.

If φ can be inferred by zero applications, then φ is already in $\mathrm{Th}(G)$ and we are done, as this corresponds to case 0 of the theorem.

Now consider an application of transitivity inferring φ of the form $u \preceq w$ from $u \preceq v \preceq w$, where, by induction, the theorem can already be assumed to hold for the inequalities $u \preceq v$ and $v \preceq w$. Since begin-variables (end-variables) never appear on the right-hand (left-hand) side of an inequality, v cannot be a begin-variable (end-variable). That leaves us with two cases, with v being either a create- or a use-variable.

Case v is a create-variable Let v be a create-variable, say create(A_v). Let us list the possibilities for u and note the relevant properties:

(a) u is also a create-variable, say create(A_u). By induction, we know that the inequality $u \preceq v$ either already belongs to $\mathrm{Th}(G)$, so there is an edge $A_v \rightarrow A_u$ in G (Axiom 4), or the inequality corresponds to case 1 of the theorem, therefore $G \vdash A_v \dashrightarrow A_u$. In either case we have $G \vdash A_v \dashrightarrow A_u$.

(b) u is a begin-variable, say begin(P_u). By induction, $u \preceq v$ either belongs to $\mathrm{Th}(G)$, so there is an edge $A_v \overset{!}{\rightarrow} P_u$ in G (Axiom 2) or $A_v \rightarrow P_u$ in G (Axiom 5); or $u \preceq v$ corresponds to case 2 of the theorem, therefore $G \vdash A_v \dashrightarrow P_u$. In either case we have $G \vdash A_v \dashrightarrow P_u$.

(c) u is a use-variable, say use(P_u, r_u, A_u). By induction, $u \preceq v$ either (c1) belongs to $\mathrm{Th}(G)$, so there is some use–generate–derive triangle (A_v, A_u, P_u, r_u) in G (Axiom 8); or (c2) $u \preceq v$ corresponds to case 7 of the theorem, thus there is a use–generate–derive triangle (A'_v, A_u, P_u, r_u) in G with $G \vdash A_v \dashrightarrow A'_v$.

We also list the possibilities for w and their relevant properties:

(d) w is also a create-variable, say create(A_w). By the induction hypothesis applied to $v \preceq w$, reasoning similarly as in case (a) above, we have $G \vdash A_w \dashrightarrow A_v$.

(e) w is an end-variable, say end(P_w). By induction, $v \preceq w$ either belongs to Th(G), so there is an edge $A_v \xrightarrow{!} P_w$ in G (Axiom 2) or $P_w \rightarrow A_v$ in G (Axiom 6); or $v \preceq w$ corresponds to case 3 of the theorem, therefore $G \vdash P_w \dashrightarrow A_v$. We have thus either (e1) $A_v \xrightarrow{!} P_w$ in G or (e2) $G \vdash P_w \dashrightarrow A_v$.

(f) w is a use-variable, say use(P_w, r_w, A_w). This necessitates the presence of edge $P_w \xrightarrow{r_w} A_w$ in G. By induction, the inequality $v \preceq w$ either (f1) belongs to Th(G), so that $A_w = A_v$ (Axiom 3); or (f2) $v \preceq w$ corresponds to case 5 of the theorem, thus $G \vdash A_w \dashrightarrow A_v$.

We can now inspect the nine possible combinations:

(ad) φ is create$(A_u) \preceq$ create(A_w). From $G \vdash A_v \dashrightarrow A_u$ and $G \vdash A_w \dashrightarrow A_v$ we infer $G \vdash A_w \dashrightarrow A_u$, which matches case 1 of the theorem.

(ae) φ is create$(A_u) \preceq$ end(P_w). From $G \vdash A_v \dashrightarrow A_u$ and either $A_v \xrightarrow{!} P_w$ in G or $G \vdash P_w \dashrightarrow A_v$ we infer $G \vdash P_w \dashrightarrow A_u$, which matches case 3 of the theorem.

(af) φ is create$(A_u) \preceq$ use(P_w, r_w, A_w) with $P_w \xrightarrow{r_w} A_w$ in G. In case f1, we have $G \vdash A_v \dashrightarrow A_u$ and $A_v = A_w$, so the case corresponds to case 5 of the theorem. In case f2, we infer $G \vdash A_w \dashrightarrow A_u$ from $G \vdash A_w \dashrightarrow A_v$ and $G \vdash A_v \dashrightarrow A_u$, which again matches case 5 of the theorem.

(bd) φ is begin$(P_u) \preceq$ create(A_w). From $G \vdash A_v \dashrightarrow P_u$ and $G \vdash A_w \dashrightarrow A_v$ we infer $G \vdash A_w \dashrightarrow P_u$, which corresponds to case 2 of the theorem.

(be) φ is begin$(P_u) \preceq$ end(P_w). From $G \vdash A_v \dashrightarrow P_u$ and either $A_v \xrightarrow{!} P_w$ in G or $G \vdash P_w \dashrightarrow A_v$ we infer $G \vdash P_w \dashrightarrow P_u$, which matches case 4 of the theorem.

(bf) φ is begin$(P_u) \preceq$ use(P_w, r_w, A_w) with $P_w \xrightarrow{r_w} A_w$ in G. In case f1, we have $G \vdash A_v \dashrightarrow P_u$ and $A_v = A_w$, so the case corresponds to case 6 of the theorem. In case f2, we infer $G \vdash A_w \dashrightarrow P_u$ from $G \vdash A_w \dashrightarrow A_v$ and $G \vdash A_v \dashrightarrow P_u$, which again matches case 6 of the theorem.

(cd) φ is use$(P_u, r_u, A_u) \preceq$ create(A_w). Case c1 corresponds directly to case 7 of the theorem. In case c2, we infer $G \vdash A_w \dashrightarrow A'_v$ from $G \vdash A_v \dashrightarrow A'_v$ and $G \vdash A_w \dashrightarrow A_v$, which again matches case 7 of the theorem.

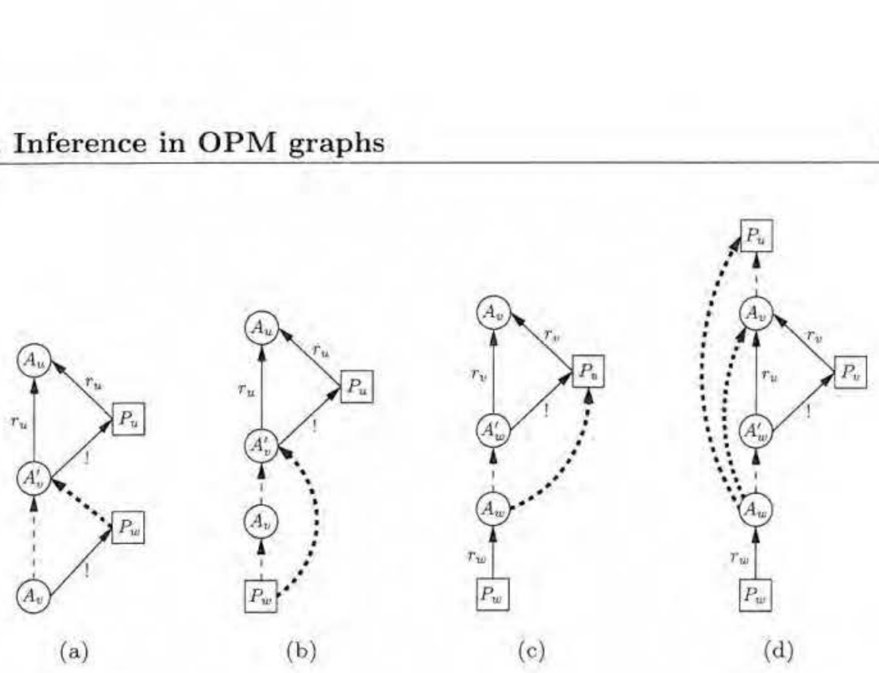


Figure 5.8: Proof of the completeness of Theorem 5.14 for cases (a) c2 with e1, (b) c2 with e2, (c) h1 with l, and (d) h2 with l. The bold edges are newly inferred.

(ce) φ is use$(P_u, r_u, A_u) \preceq$ end(P_w). First, consider case c1 together with e1. Since $G \bigtriangleup (A_v, A_u, P_u, r_u)$, P_u and P_w must be equal because G is legal. In this case the inequality holds by Axiom 3. Case c1 together with e2 corresponds directly to case 8 of the theorem. Finally, in case c2, from $G \vdash A_v \dashrightarrow A'_v$, and either $A_v \xrightarrow{!} P_w$ (from e1, see Figure 5.8(a)) or $G \vdash P_w \dashrightarrow A_v$ (from e2, see Figure 5.8(b)) we infer $G \vdash P_w \dashrightarrow A'_v$, which matches case 8 of the theorem.

(cf) φ is use$(P_u, r_u, A_u) \preceq$ use(P_w, r_w, A_w). Case c1 together with f1 corresponds directly to case 9a of the theorem. Case c1 together with f2 matches case 9b of the theorem. The same holds for c2 together with f1. In case c2 together with f2 we infer $G \vdash A_w \dashrightarrow A'_v$ from $G \vdash A_w \dashrightarrow A_v$ and $G \vdash A_v \dashrightarrow A'_v$, which again matches case 9b of the theorem.

Case v is a use-variable

Let v be a use-variable, say use(P_v, r_v, A_v). Note that this necessitates the presence of the edge $P_v \xrightarrow{r_v} A_v$ in G. Let us list the possibilities for u and note the relevant properties:

(g) u is a create-variable, say create(A_u). By induction, we know that the inequality $u \preceq v$ either (g1) already belongs to Th(G), so A_u equals A_v with the edge $P_v \xrightarrow{r_v} A_v$ in G (Axiom 3); or (g2) the inequality corresponds to case 5 of the theorem, therefore $G \vdash A_v \dashrightarrow A_u$ with the edge $P_v \xrightarrow{r_v} A_v$ in G.

(h) u is a begin-variable, say begin(P_u). By induction, $u \preceq v$ either (h1) already belongs to Th(G), thus P_u equals P_v with the edge $P_v \xrightarrow{r_v} A_v$ in G (Axiom 3); or (h2) the inequality corresponds to case 6 of the theorem, therefore $G \vdash A_v \dashrightarrow P_u$ with the edge $P_v \xrightarrow{r_v} A_v$ in G.

(i) u is also a use-variable, say use(P_u, r_u, A_u). By induction, we know that the inequality $u \preceq v$ can only correspond to case 9 of the theorem, therefore we have some triangle (A'_v, A_u, P_u, r_u) in G with the edge $P_v \xrightarrow{r_v} A_v$ in G, and either $A_v = A'_v$ or $G \vdash A_v \dashrightarrow A'_v$.

We also list the possibilities for w and their relevant properties:

(j) w is a create-variable, say create(A_w). By induction, $v \preceq w$ either (j1) already belongs to Th(G), so there is some use–generate–derive triangle (A_w, A_v, P_v, r_v) in G (Axiom 8); or (j2) the inequality corresponds to case 7 of the theorem, and there is a use–generate–derive triangle (A'_w, A_v, P_v, r_v) in G with $G \vdash A_w \dashrightarrow A'_w$. Note that in both cases we can infer $G \vdash A_w \dashrightarrow A_v$. Indeed, in case j1 we have the edge $A_w \xrightarrow{r_v} A_v$ in G. In case j2 we have $G \vdash A_w \dashrightarrow A'_w$ and the edge $A'_w \xrightarrow{r_v} A_v$ in G.

(k) w is an end-variable, say end(P_w). By induction, $v \preceq w$ either (k1) already belongs to Th(G), so P_w equals P_v with the edge $P_v \xrightarrow{r_v} A_v$ in G (Axiom 3); or (k2) corresponds to case 8 of the theorem, thus there is some use–generate–derive triangle (A_w, A_v, P_v, r_v) in G with $G \vdash P_w \dashrightarrow A_w$. Note that in both cases we can infer $G \vdash P_w \dashrightarrow A_v$. Indeed, in case k1 we have the edge $P_w \xrightarrow{r_v} A_v$ in G. In case k2 we have $G \vdash P_w \dashrightarrow A_w$ and the edge $A_w \xrightarrow{r_v} A_v$ in G.

(l) w is also a use-variable, say use(P_w, r_w, A_w). By induction, we know that the inequality $v \preceq w$ can only correspond to case 9 of the theorem, so there is some use–generate–derive triangle (A'_w, A_v, P_v, r_v) in G with $P_w \xrightarrow{r_w} A_w$ in G, and either $A_w = A'_w$ or $G \vdash A_w \dashrightarrow A'_w$. Note that in both cases we can infer $G \vdash A_w \dashrightarrow A_v$ by the edge $A'_w \xrightarrow{r_v} A_v$ in the triangle.

We can now inspect the nine possible combinations:

(gj) φ is create$(A_u) \preceq$ create(A_w). In case g1, we have $G \vdash A_w \dashrightarrow A_v$ and $A_v = A_u$, so the case corresponds directly to case 1 of the theorem. In case g2, we have $G \vdash A_w \dashrightarrow A_v$ and $G \vdash A_v \dashrightarrow A_u$, so we can infer $G \vdash A_w \dashrightarrow A_u$, which, again, matches case 1 of the theorem.

(gk) φ is create$(A_u) \preceq$ end(P_w). In case g1 together with k1, we have $A_v = A_u$, $P_v = P_w$, and the edge $P_v \xrightarrow{r_v} A_v$ in G, so the case corresponds directly to case 3 of the theorem. In case g2 together with k1, we have $P_v = P_w$ with the edge $P_v \xrightarrow{r_v} A_v$ in G. From the latter and $G \vdash A_v \dashrightarrow A_u$, we infer $G \vdash P_w \dashrightarrow A_u$, which again matches case 3 of the theorem. In case g1 together with k2, we have $G \vdash P_w \dashrightarrow A_v$ and $A_v = A_u$, so the case corresponds directly to case 3 of the theorem. In case g2 together with k2, we infer $G \vdash P_w \dashrightarrow A_u$ from $G \vdash P_w \dashrightarrow A_v$ and $G \vdash A_v \dashrightarrow A_u$. Hence the case again matches case 3 of the theorem.

(gl) φ is create$(A_u) \preceq$ use(P_w, r_w, A_w). In case g1, we have $G \vdash A_w \dashrightarrow A_v$ and $A_v = A_u$, so the case corresponds directly to case 5 of the theorem. In case g2, we infer $G \vdash A_w \dashrightarrow A_u$ from $G \vdash A_w \dashrightarrow A_v$ and $G \vdash A_v \dashrightarrow A_u$, which matches case 5 of the theorem.

(hj) φ is begin$(P_u) \preceq$ create(A_w). By case j, we infer $G \vdash A_w \dashrightarrow P_v$. (Indeed, in case j1 we easily infer $G \vdash A_w \dashrightarrow P_v$. In case j2 we also infer $G \vdash A_w \dashrightarrow P_v$ from $G \vdash A_w \dashrightarrow A'_w$ and $A'_w \xrightarrow{!} P_v$ in G.) Now in case h1 we have $P_v = P_u$, so the case corresponds directly to case 2 of the theorem. In case h2 we have $G \vdash A_w \dashrightarrow A_v$ and $G \vdash A_v \dashrightarrow P_u$, so we can infer $G \vdash A_w \dashrightarrow P_u$. Thus the case again matches case 2 of the theorem.

(hk) φ is begin$(P_u) \preceq$ end(P_w). In case h1 together with k1, we have $P_u = P_v = P_w$, so the inequality trivially holds (Axiom 1). In case h1 together with k2, we have $P_v = P_u$, and we infer $G \vdash P_w \dashrightarrow P_v$ from $G \vdash P_w \dashrightarrow A_w$ and $G \vdash A_w \dashrightarrow P_v$. Thus the case corresponds to case 4 of the theorem. In case h2 we have $G \vdash P_w \dashrightarrow A_v$, which combined with $G \vdash A_v \dashrightarrow P_u$, yields $G \vdash P_w \dashrightarrow P_u$. Hence the case again matches case 4 of the theorem.

(hl) φ is begin$(P_u) \preceq$ use(P_w, r_w, A_w). By case l, we infer $G \vdash A_w \dashrightarrow P_v$ from $A'_w \xrightarrow{!} P_v$ in G, and either $A_w = A'_w$ or $G \vdash A_w \dashrightarrow A'_w$. Also, there is an edge $P_w \xrightarrow{r_w} A_w$ in G, and $G \vdash A_w \dashrightarrow A_v$. In case h1 (see Figure 5.8(c)) we have $P_v = P_u$, so the case corresponds to case 6 of the theorem. In case h2 (see Figure 5.8(d)), from $G \vdash A_w \dashrightarrow A_v$ and $G \vdash A_v \dashrightarrow P_u$, we infer $G \vdash A_w \dashrightarrow P_u$. Hence the case again matches case 6 of the theorem.

(ij) φ is use$(P_u, r_u, A_u) \preceq$ create(A_w). We infer $G \vdash A_w \dashrightarrow A'_v$ from

$G \vdash A_w \dashrightarrow A_v$, and either $A_v = A'_v$ or $G \vdash A_v \dashrightarrow A'_v$. Together with $G \triangle (A'_v, A_u, P_u, r_u)$, the case corresponds to case 7 of the theorem.

(ik) φ is $\mathrm{use}(P_u, r_u, A_u) \preceq \mathrm{end}(P_w)$. We already have the triangle $G \triangle (A'_v, A_u, P_u, r_u)$. In case k1, we have $P_w = P_v$. We infer $G \vdash P_v \dashrightarrow A'_v$ from the edge $P_v \xrightarrow{r_v} A_v$ in G, and either $A_v = A'_v$ or $G \vdash A_v \dashrightarrow A'_v$. The case thus matches case 8 of the theorem. In case k2, we infer $G \vdash P_w \dashrightarrow A'_v$ from $G \vdash P_w \dashrightarrow A_w$, $A_w \xrightarrow{r_v} A_v$ in G, and either $A_v = A'_v$ or $G \vdash A_v \dashrightarrow A'_v$. Hence the case again matches case 8 of the theorem.

(il) φ is $\mathrm{use}(P_u, r_u, A_u) \preceq \mathrm{use}(P_w, r_w, A_w)$. We already have $P_w \xrightarrow{r_w} A_w$ in G and $G \triangle (A'_v, A_u, P_u, r_u)$. We additionally infer $G \vdash A_w \dashrightarrow A'_v$ from $G \vdash A_w \dashrightarrow A_v$, and either $A_v = A'_v$ or $G \vdash A_v \dashrightarrow A'_v$. Hence the case matches case 9b of the theorem.

5.3.4 About no-use inequalities

The OPM reference specification [MCF$^+$11] defines edges adjacent to artifacts in terms of the creation of the artifact, with respect to the creation of another artifact, or the beginning and ending of a process. There is some value in considering a temporal theory that ignores use time-points, since the theory becomes simpler (though it is unable to tell us anything about usage of artifacts). In this case, it is worth characterising temporal inference in the context of this simpler theory.

First we state a remarkable corollary, after introducing the following definition.

Definition 5.15. If in an inequality φ of the form $u \preceq v$, neither u nor v is a use-variable, then we call φ a *no-use* inequality.

As a corollary to Theorem 5.14, we obtain the following completeness result for edge inference, as far as no-use inequalities are concerned. Note that the edge-inference rules are present as Rules 1–4 in Figure 5.7.

Corollary 5.16. *Let G be a legal OPM graph and let φ be a no-use inequality. Then $\mathrm{Th}(G) \models \varphi$ if and only if φ can be inferred using Axioms 1–2 and Rules 1–4 in Figure 5.7.*

Proof. It is clear from Theorem 5.14 that if φ can be inferred using Axioms 1–2 and Rules 1–4, then $\mathrm{Th}(G) \models \varphi$. For the other direction, assume that $\mathrm{Th}(G) \models \varphi$ holds. Then we know by Theorem 5.14 that φ can be inferred by the axioms and rules presented in Figure 5.7. By examination of these axioms and rules, however, we notice that Axioms 1–2 and Rules 1–4 are the only ones that infer no-use inequalities. □

It is interesting to note, that when dealing with no-use inequalities, we do not need the full temporal theory of an OPM graph. We start with a small generalisation of Definitions 5.8 and 5.11.

Definition 5.17. Let G be a legal OPM graph and let $u, v \in \mathit{Vars}(G)$. Let Σ be a subset of $\mathrm{Th}(G)$. Any temporal interpretation that satisfies all inequalities of Σ is called a *temporal model* of Σ. Furthermore, the inequality $u \preceq v$ is a *logical consequence* of Σ, denoted by $\Sigma \models u \preceq v$, if $u \preceq v$ is satisfied in every temporal model of Σ.

We can now select, for a given OPM graph G, only the no-use inequalities from its temporal theory.

Definition 5.18. For a legal OPM graph G, we define the *no-use temporal theory* of G, denoted by $\mathrm{Th}^{\text{no-use}}(G)$, as follows:

$$\begin{aligned}\mathrm{Th}^{\text{no-use}}(G) = \{\varphi \in \mathrm{Th}(G) \mid \varphi \text{ is a no-use inequality}\} \\ \cup \left\{\mathrm{create}(A) \preceq \mathrm{end}(P) \mid P \xrightarrow{!} A \text{ in } G\right\} \\ \cup \left\{\mathrm{create}(B) \preceq \mathrm{create}(A) \mid A \xrightarrow{!} B \text{ in } G\right\}.\end{aligned}$$

The intuition is that $\mathrm{Th}^{\text{no-use}}(G)$ does not contain Axioms 3 and 8, and enforces Axioms 4 and 6 for precise and imprecise edges alike.[§]

We can now observe that use-variables do not influence the no-use inequalities that are logical consequences of $\mathrm{Th}(G)$.

Proposition 5.19. *Let G be a legal OPM graph and let φ be a no-use inequality. Then $\mathrm{Th}(G) \models \varphi$ if and only if $\mathrm{Th}^{\text{no-use}}(G) \models \varphi$.*

Proof. Since any temporal model τ of $\mathrm{Th}(G)$ is also a temporal model of $\mathrm{Th}^{\text{no-use}}(G)$, the if-direction is immediate. For the only-if direction, let τ be a temporal model of $\mathrm{Th}^{\text{no-use}}(G)$. We try to extend τ to τ' in such a way that τ' is a temporal model of $\mathrm{Th}(G)$. For every no-use variable u simply put $\tau'(u) = \tau(u)$. Now we have to find suitable values for:

[§]Note that in the full theory $\mathrm{Th}(G)$, the no-use inequality $\mathrm{create}(A) \preceq \mathrm{end}(P)$ for $P \xrightarrow{!} A$ in G is implied by Axiom 3, but since we omit this axiom, we need to recover the inequality in Axiom 6. Likewise, the no-use inequality $\mathrm{create}(B) \preceq \mathrm{create}(A)$ for $A \xrightarrow{!} B$ in G is provided by Lemma 5.12. Since the proof of the lemma utilises use-variables, the lemma doesn't hold anymore and we need to recover the inequality in Axiom 4.

- use(P, r, A) for each $P \xrightarrow{r} A$ in G, so that Axiom 3 is satisfied, and
- use(P, r, B) for each $G \triangle (A, B, P, r)$, so that Axiom 8 is satisfied.

It is easy to verify that Axiom 3 holds if $\tau'(\text{use}(P, r, A))$ equals the maximum of $\tau(\text{create}(A))$ and $\tau(\text{begin}(P))$. Likewise, Axiom 8 holds if $\tau'(\text{use}(P, r, B))$ equals the maximum of $\tau(\text{create}(B))$ and $\tau(\text{begin}(P))$. Thus τ' satisfies all eight axioms and is a temporal model of Th(G). We know thus that τ' satisfies φ. Since φ is a no-use inequality, and τ' and τ coincide on all variables used in no-use inequalities, τ also satisfies φ. □

The above proposition together with Corollary 5.16 yields the following:

Corollary 5.20. *Let G be a legal OPM graph and let φ be a no-use inequality. Then* $\text{Th}^{\textit{no-use}}(G) \models \varphi$ *if and only if φ can be inferred using Axioms 1–2 and Rules 1–4 in Figure 5.7.*

This section provides a remarkable result since it establishes the completeness of edge inferences (Rules 1–4 in Figure 5.7) for no-use inequalities. Furthermore, reasoning with use time-points does not allow us to derive any new inequality about no-use variables. We envisage this result to be leveraged by developers of reasoners for OPM, since it offers opportunities to optimize reasoners, by reducing the number of time-points to reason over, focusing on no-use variables in a first phase, and dealing efficiently with use-variables afterwards.

5.4 Operations on OPM graphs

The reason for capturing provenance is that it can be used to address a variety of use cases [MGBM07]. To this end, one needs to collect provenance information from potentially different sources, combine and process it in multiple ways. It is therefore useful to define operations on OPM graphs, which we anticipate can become part of "provenance toolkits".

When two OPM graphs are obtained from different sources, a reasoner may want to take their union, if it ascertains they relate to some common entities. Given two OPM graphs, an intersection operation helps identify their common elements. Different sources may use different identifiers for graph nodes; thus, to be able to compute meaningful union and intersection, it may be required to rename some nodes, before performing these operations. In this section, we formalise notions of subgraph, union, intersection, and renaming and merging.

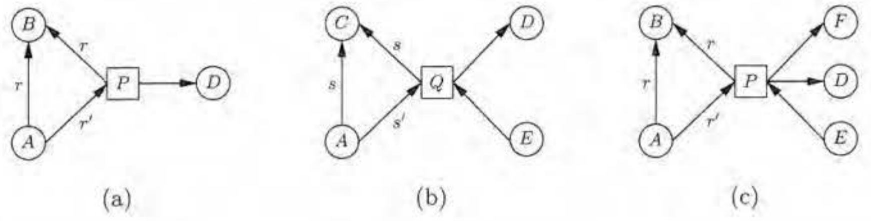


Figure 5.9: Three legal OPM graphs.

Let us fix two OPM graphs G and H for use in this section. Neither G nor H have to be legal.

Definition 5.21 (Subgraph). H is a *subgraph* of G if every constituent of H is a subset of the corresponding constituent of G. Formally:

- $Art^H \subseteq Art^G$,
- $Proc^H \subseteq Proc^G$,
- $Roles^H \subseteq Roles^G$,
- $GeneratedBy!^H \subseteq GeneratedBy!^G$,
- $Used!^H \subseteq Used!^G$,
- $DerivedFrom!^H \subseteq DerivedFrom!^G$,
- $GeneratedBy^H \subseteq GeneratedBy^G$,
- $Used^H \subseteq Used^G$,
- $DerivedFrom^H \subseteq DerivedFrom^G$,
- $InformedBy^H \subseteq InformedBy^G$.

Note that a subgraph of a legal OPM graph may not be legal. For example, the graph presented in Figure 5.9(a) is legal, whereas its subgraph composed of nodes A, B, P, role r, and edges $A \xrightarrow{r} B$ and $A \xrightarrow{r'} P$ is not legal, since the use–generate–derive triangle (A, B, P, r) is not complete.

Definition 5.22 (Union). The *union* of G and H, denoted by $G \cup H$, is the OPM graph where each constituent equals the union of the corresponding constituents in G and H. Formally:

- $Art^{G \cup H} = Art^G \cup Art^H$,

- $\mathit{Proc}^{G \cup H} = \mathit{Proc}^G \cup \mathit{Proc}^H$,
- $\mathit{Roles}^{G \cup H} = \mathit{Roles}^G \cup \mathit{Roles}^H$,
- $\mathit{GeneratedBy}!^{G \cup H} = \mathit{GeneratedBy}!^G \cup \mathit{GeneratedBy}!^H$,
- $\mathit{Used}!^{G \cup H} = \mathit{Used}!^G \cup \mathit{Used}!^H$,
- $\mathit{DerivedFrom}!^{G \cup H} = \mathit{DerivedFrom}!^G \cup \mathit{DerivedFrom}!^H$,
- $\mathit{GeneratedBy}^{G \cup H} = \mathit{GeneratedBy}^G \cup \mathit{GeneratedBy}^H$,
- $\mathit{Used}^{G \cup H} = \mathit{Used}^G \cup \mathit{Used}^H$,
- $\mathit{DerivedFrom}^{G \cup H} = \mathit{DerivedFrom}^G \cup \mathit{DerivedFrom}^H$,
- $\mathit{InformedBy}^{G \cup H} = \mathit{InformedBy}^G \cup \mathit{InformedBy}^H$.

Note that the union of two legal OPM graphs may not be legal. For instance, the graph presented in Figure 5.9(a) is legal, and so is the graph in Figure 5.9(b). The union of these two graphs, however, is not legal, since in the union A has two different precise generated-by edges: $A \xrightarrow{r'} P$ and $A \xrightarrow{s'} Q$.

Definition 5.23 (Intersection). The *intersection* of G and H, denoted by $G \cap H$, is the OPM graph where each constituent equals the intersection of the corresponding constituents in G and H. Formally:

- $\mathit{Art}^{G \cap H} = \mathit{Art}^G \cap \mathit{Art}^H$,
- $\mathit{Proc}^{G \cap H} = \mathit{Proc}^G \cap \mathit{Proc}^H$,
- $\mathit{Roles}^{G \cap H} = \mathit{Roles}^G \cap \mathit{Roles}^H$,
- $\mathit{GeneratedBy}!^{G \cap H} = \mathit{GeneratedBy}!^G \cap \mathit{GeneratedBy}!^H$,
- $\mathit{Used}!^{G \cap H} = \mathit{Used}!^G \cap \mathit{Used}!^H$,
- $\mathit{DerivedFrom}!^{G \cap H} = \mathit{DerivedFrom}!^G \cap \mathit{DerivedFrom}!^H$,
- $\mathit{GeneratedBy}^{G \cap H} = \mathit{GeneratedBy}^G \cap \mathit{GeneratedBy}^H$,
- $\mathit{Used}^{G \cap H} = \mathit{Used}^G \cap \mathit{Used}^H$,
- $\mathit{DerivedFrom}^{G \cap H} = \mathit{DerivedFrom}^G \cap \mathit{DerivedFrom}^H$,
- $\mathit{InformedBy}^{G \cap H} = \mathit{InformedBy}^G \cap \mathit{InformedBy}^H$.

Note that G and H need to have at least one node or one role in common for their intersection to be non-empty. For example, the intersection of the two graphs in Figures 5.9(a) and 5.9(b) yields the OPM graph consisting of the artifacts A and D.

The following is readily verified:

Proposition 5.24. *The intersection of two legal OPM graphs is legal.*

One may wonder about the relations between union and intersection of legal OPM graphs and their temporal theories. We answer this question next. Let G and H now be two legal OPM graphs.

Proposition 5.25. $\mathrm{Th}(G \cup H) = \mathrm{Th}(G) \cup \mathrm{Th}(H)$.

Proof. Each inequality in $\mathrm{Th}(G)$ or $\mathrm{Th}(H)$ corresponds to a single node, a single edge or some use–generate–derive triangle present in G or H. Thus all inequalities present in $\mathrm{Th}(G) \cup \mathrm{Th}(H)$, also belong to $\mathrm{Th}(G \cup H)$. Moreover, the only additional inequalities in $\mathrm{Th}(G \cup H)$ would correspond to some use–generate-derive triangles that were newly formed by the union of G and H. Since both G and H are legal, this is impossible, because legal OPM graphs cannot contain parts of a use–generate-derive triangle. Therefore, $\mathrm{Th}(G \cup H)$ contains only inequalities that are already present in $\mathrm{Th}(G)$, or in $\mathrm{Th}(H)$, or in both. □

Proposition 5.26. $\mathrm{Th}(G \cap H) \subseteq \mathrm{Th}(G) \cap \mathrm{Th}(H)$.

Proof. Any inequality from $\mathrm{Th}(G \cap H)$ corresponds to a single node, an edge, or a use–generate–derive triangle present in $G \cap H$, and thus in both G and H. Therefore, it also belongs to $\mathrm{Th}(G) \cap \mathrm{Th}(H)$. □

The converse inclusion does not hold. If G consists only of edge $P \rightarrow A$, and H consists only of edge $A \xrightarrow{!} P$, then $G \cap H$ consists of the two nodes A and P. So

$$\mathrm{Th}(G) = \{\mathrm{create}(A) \preceq \mathrm{end}(P), \mathrm{begin}(P) \preceq \mathrm{end}(P)\} ,$$
$$\mathrm{Th}(H) = \{\mathrm{begin}(P) \preceq \mathrm{create}(A), \mathrm{create}(A) \preceq \mathrm{end}(P), \mathrm{begin}(P) \preceq \mathrm{end}(P)\} ,$$

and

$$\mathrm{Th}(G \cap H) = \{\mathrm{begin}(P) \preceq \mathrm{end}(P)\} .$$

Clearly $\mathrm{create}(A) \preceq \mathrm{end}(P) \in \mathrm{Th}(G) \cap \mathrm{Th}(H) \not\subseteq \mathrm{Th}(G \cap H)$. Note that $\mathrm{create}(A) \preceq \mathrm{end}(P)$ is not even a logical consequence of $\mathrm{Th}(G \cap H)$.

5.4.1 Renaming and merging

By definition, the nodes and roles of an OPM graph are local to the graph. Prior to performing a union or an intersection of two OPM graphs G and H, we may need to resolve some identity issues between the nodes and roles in the graphs. For example, some process node P in G may represent the same actual process as some process node Q in H. Likewise, role r in edge $P \xrightarrow{r} B$ in G may refer to the same actual role as role s in edge $Q \xrightarrow{s} C$ in H, and also B and C may represent the same actual artifact. Moreover, it is equally possible that some node or role is accidentally used in both graphs whereas this node or role does *not* represent the same actual entity across the two graphs. Resolving such identity issues leads to a renaming operation on one or both of the graphs, whereby nodes and roles representing the same actual entity can be renamed to a common node or role; likewise, nodes and roles not representing the same actual entity, but accidentally used in both graphs, can be renamed to distinct nodes or roles.

Definition 5.27 (Renaming). Let G and H be OPM graphs, which need not be legal. Let ρ_{Art} be a bijection from Art^G to a finite set Art', let ρ_{Proc} be a bijection from $Proc^G$ to a finite set $Proc'$, and let ρ_{Roles} be a bijection from $Roles^G$ to a finite set $Roles'$, with the sets Art', $Proc'$, and $Roles'$ mutually disjoint. Then H is the *renaming* of G by ρ_{Art}, ρ_{Proc}, and ρ_{Roles}, if the following holds:

- $Art^H = Art'$,
- $Proc^H = Proc'$,
- $Roles^H = Roles'$,
- $GeneratedBy!^H = \{(\rho_{Art}(A), \rho_{Roles}(r), \rho_{Proc}(P)) \mid (A, r, P) \in GeneratedBy!^G\}$;
- $Used!^H = \{(\rho_{Proc}(P), \rho_{Roles}(r), \rho_{Art}(A)) \mid (P, r, A) \in Used!^G\}$;
- $DerivedFrom!^H = \{(\rho_{Art}(A), \rho_{Roles}(r), \rho_{Art}(B)) \mid (A, r, B) \in DerivedFrom!^G\}$;
- $GeneratedBy^H = \{(\rho_{Art}(A), \rho_{Proc}(P)) \mid (A, P) \in GeneratedBy^G\}$;
- $Used^H = \{(\rho_{Proc}(P), \rho_{Art}(A)) \mid (P, A) \in Used^G\}$;
- $DerivedFrom^H = \{(\rho_{Art}(A), \rho_{Art}(B)) \mid (A, B) \in DerivedFrom^G\}$;
- $InformedBy^H = \{(\rho_{Proc}(P), \rho_{Proc}(Q)) \mid (P, Q) \in InformedBy^G\}$;

Note that Art^G and Art' need not be disjoint; similarly, neither $Proc^G$ and $Proc'$, nor $Roles^G$ and $Roles'$, need to be disjoint. Indeed, ρ_{Art}, ρ_{Proc} and ρ_{Roles} may coincide with the identity function on some of their inputs, i.e., not all artifacts, processes and roles need to be renamed.

Example 5.28. We can rename the graph presented in Figure 5.9(b) by the following bijections:

- $\rho_{Art}(A) = A$, $\rho_{Art}(C) = B$, $\rho_{Art}(D) = F$, and $\rho_{Art}(E) = E$;
- $\rho_{Proc}(Q) = P$;
- $\rho_{Roles}(s) = r$ and $\rho_{Roles}(s') = r'$.

Then we can take the union of the renamed graph with the graph shown in Figure 5.9(a), which yields the legal OPM graph presented in Figure 5.9(c). □

The following is readily verified:

Proposition 5.29. *The renaming of a legal OPM graph is legal.*

We next define the following generalisation of renaming.

Definition 5.30 (Merge-renaming). Let G and H be OPM graphs, which need not be legal. Let ρ_{Art}, ρ_{Proc} and ρ_{Roles} be as in Definition 5.27 except that ρ_{Art}, ρ_{Proc} and ρ_{Roles} need not be bijective: they only need to be surjective (onto) mappings. Then we say that H is the *merge-renaming* of G by ρ_{Art}, ρ_{Proc}, and ρ_{Roles}, exactly if the same equalities of Definition 5.27 hold.

Merge-renaming allows the coalescing of two or more nodes to a single node (or two or more roles to a single role). Coalescing of nodes or roles may be performed when analysing an OPM graph on a coarser level of detail. But coalescing may also be practical when more information becomes available. For example, in a traffic accident scenario, there may be observations about a "blue car" and other observations about a "Toyota", only to realise later that the blue car is the Toyota.

In contrast to Proposition 5.29, the merge-renaming of a legal OPM graph need not be legal. For example, in Figure 5.10(c), if we coalesce C and D into a single artifact E, but do not coalesce P and Q, nor their roles, then E has two distinct precise generated-by edges.

As a merge-renaming can coalesce artifacts, such an operation can introduce cycles of derived-from edges to an OPM graph. In the next section, we investigate the consequences of such cycles in OPM graphs.

5.4.2 Equality inference

Our definitions allow the presence of derived-from cycles in legal OPM graphs. By a *derived-from cycle*, we mean a directed simple cycle composed of derived-from edges (precise or imprecise). An OPM graph resulting from a typical experimental provenance collection procedure does not contain such cycles, and indeed the current OPM reference specification forbids them. For example, it would be strange to assert that A is derived from B and that B is derived from A.

Nevertheless, cycles may arise in a graph when, after a merge operation, certain nodes coalesce. Suppose, for example, that we have three artifacts $A \to B \to C$ without a cycle. If an application does not need the full level of detail provided, it may consider, for example, A and C to be the same at a coarser level of detail. As a consequence, a cycle $A \to B \to C = A$ is created.

Thus, we do not want to disallow derived-from cycles in OPM graphs from the outset. It is important, however, to understand the consequences of the presence of such cycles. We observe that they enforce the equality of certain temporal variables. In the preceding example, we would have $\text{create}(A) = \text{create}(B) = \text{create}(C)$.

First of all, we point out that every OPM graph has a trivial model τ_{triv} consisting of a single time-point t_0 with $\tau_{\text{triv}}(u) = t_0$ for every temporal variable u. Indeed, since the temporal theory of an OPM graph consists only of non-strict inequalities, this interpretation trivially satisfies all non-strict inequalities. Of course, that does not mean that this trivial model is the *only* model that the OPM graph possesses. On the contrary: intuitively, on a fine enough temporal granularity, we should expect that every OPM graph indeed possesses a model where all temporal variables are assigned *distinct* time-points. We observe that this is indeed always possible provided there are no derived-from cycles.

Formally, we fix some OPM graph G for this section. Let us say that a temporal interpretation $(T, \leq, \tau)$ of G is *all-distinct* if $\tau(u) \neq \tau(v)$ for any two distinct temporal variables u and v of G. When, in addition, $\leq$ is a total order on T, we say that τ has the *strict linear order* property.

Proposition 5.31. *If G does not contain any derived-from cycles, then G has an all-distinct temporal model that even satisfies the strict-linear-order property.*

Proof. We construct a total order on all temporal variables of G that is a temporal model of G under the identity mapping. Since G does not contain any derived-from cycles, we can linearly order all artifacts so that no artifact has a derived-from edge from an artifact coming later in the order. Note that there may be many possibilities for such an ordering. Any such ordering imposes an ordering on the corresponding create-variables. All begin-variables are placed before all create-variables, in some arbitrary order among them, and similarly all end-variables are placed after all create-variables. Finally, a use-variable involving artifact A is placed immediately as a successor of create(A). If there are more than one use-variables for the same artifact A, they can all be placed in an arbitrary order right after create(A). By inspecting the axioms we see that this order satisfies all axioms. □

We note that Proposition 5.31 does not state that all temporal models of cycle-free graphs are all-distinct. Rather, it establishes that one such all-distinct model exists. The next proposition provides a partial converse to Proposition 5.31:

Proposition 5.32. *If G does contain a derived-from cycle of length at least two, then G cannot have an all-distinct temporal model.*

Proof. Consider a derived-from cycle and let A and B be two distinct artifacts on that cycle. Then any temporal model τ should satisfy $\tau(\text{create}(A)) \leq \tau(\text{create}(B))$ as well as $\tau(\text{create}(B)) \leq \tau(\text{create}(A))$, so $\tau(\text{create}(A))$ equals $\tau(\text{create}(B))$. Hence τ is not all-distinct. □

Propositions 5.31–5.32 do not specify what happens when there are only derived-from cycles of length one (self-loops). Moreover, for a temporal model τ, they do not specify which distinct variables u and v *cannot* have distinct $\tau(u)$ and $\tau(v)$ if the graph contains derived-from cycles. The next proposition fills these gaps by characterising exactly when two temporal variables must be equal in all temporal models.

Naturally, for two distinct temporal variables u and v of G, we write $\text{Th}(G) \models u = v$ to denote that both $\text{Th}(G) \models u \preceq v$ and $\text{Th}(G) \models v \preceq u$. Thus, if $\text{Th}(G) \models u = v$, then there is no model of G that is all-distinct since any temporal model τ must satisfy $\tau(u) \leq \tau(v)$ and $\tau(v) \leq \tau(u)$; so $\tau(u) = \tau(v)$. Intuitively, two temporal variables u and v of G such that $\text{Th}(G) \models u = v$ can be seen as indistinguishable in the given temporal model, for example as a result of coalescing some nodes in a graph with a more detailed temporal model.

Proposition 5.33. $\text{Th}(G) \models u = v$ *if and only if u and v match one of the following possibilities:*

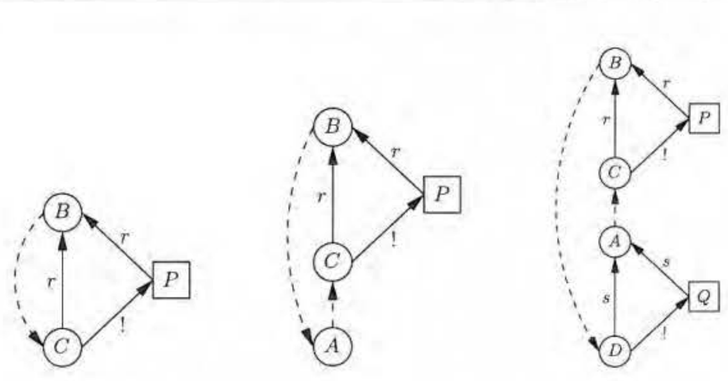


Figure 5.10: Graph patterns for Proposition 5.33.

(1) u is create(A), *v is* create(B), *and A and B belong to the same derived-from cycle.*

(2) u is create(B), *v is* use(P, r, B), *and in G, the nodes P and B, together with some node C, match the pattern shown in Figure 5.10(a). Note that B and C need not be distinct.*

(3) u is create(A), *v is* use(P, r, B), *and in G, the nodes P, A and B, together with some node C, match the pattern shown in Figure 5.10(b). Note that A, B and C need not be distinct.*

(4) u is use(P, r, B), *v is* use(Q, s, A), *and in G, nodes P, Q, A and B, together with some nodes C and D, match the pattern shown in Figure 5.10(c). Note that A, B, C and D need not be distinct, nor P and Q.*¶

Proof. The proof of the if-direction amounts to an inspection of involved patterns to verify that $\mathrm{Th}(G) \models u = v$ indeed holds. For example, let us examine pattern in Figure 5.10(c) in case 4. By Axiom 3 we have $\mathrm{Th}(G) \models \mathrm{create}(B) \preceq \mathrm{use}(P, r, B)$ for edge $P \xrightarrow{r} B$ and $\mathrm{Th}(G) \models \mathrm{create}(A) \preceq \mathrm{use}(Q, s, A)$ for edge $Q \xrightarrow{s} A$. For the triangle (C, B, P, r) and $G \vdash B \dashrightarrow C$, we can apply Rule 7 from Figure 5.7, so we have $\mathrm{Th}(G) \models \mathrm{use}(P, r, B) \preceq \mathrm{create}(B)$, and thus $\mathrm{Th}(G) \models \mathrm{create}(B) = \mathrm{use}(P, r, B)$. Likewise, for the triangle (D, A, Q, s) and $G \vdash A \dashrightarrow D$, we obtain $\mathrm{Th}(G) \models \mathrm{create}(A) = \mathrm{use}(Q, s, A)$. Since A and B belong to the same derived-from cycle, we also have $\mathrm{Th}(G) \models \mathrm{create}(A) = \mathrm{create}(B)$, hence $\mathrm{Th}(G) \models \mathrm{use}(P, r, B) = \mathrm{use}(Q, s, A)$.

¶Note that in Figure 5.10(c), if C and D coincide then $C \xrightarrow{!} P$ and $D \xrightarrow{!} Q$ must also coincide for G to be legal.

The proof of the only-if direction amounts to a lengthy but straightforward inspection of the possible cases where both $\mathrm{Th}(G) \models u \preceq v$ and $\mathrm{Th}(G) \models v \preceq u$, in the characterisation of temporal inference provided by Figure 5.7. For example, in case 4, we clearly see that we can only obtain $\mathrm{Th}(G) \models \mathrm{use}(P, r, B) = \mathrm{use}(Q, s, A)$ by combining the following rules of Figure 5.7: Rule 9a with again Rule 9a resulting in the pattern from Figure 5.10(c) with $A = C$ and $B = D$; Rule 9a with Rule 9b resulting in the pattern from Figure 5.10(c) with $A = C$; and Rule 9b with again Rule 9b resulting in the pattern from Figure 5.10(c). □

Observe that a graph that matches Figure 5.10(c) also matches Figure 5.10(b), since $B \dashrightarrow A$ can be inferred from $B \dashrightarrow D$ and $D \xrightarrow{s} A$ in Figure 5.10(c). Likewise, a graph that matches Figure 5.10(b) also matches Figure 5.10(a), since $B \dashrightarrow C$ can be inferred from $B \dashrightarrow A$ and $A \dashrightarrow C$ in Figure 5.10(b).

Hence, by repeated application of Proposition 5.33 (1)–(4), we derive that all use and create time-points for artifacts A, B, C, D in Figure 5.10(c) are equal. Likewise, we note the equality of all use and create time-points for artifacts A, B, C in Figure 5.10(b) and for B, C in Figure 5.10(a).

The OPM reference specification does not allow derived-from cycles, but it does not define a merge operation either. This section has demonstrated that a merge operation can introduce derived-from loops into OPM graphs, but they must satisfy some constraints: all time-points of artifacts involved in a loop must coalesce to a single time-point. Were a merge operation added to the reference specification, two options are possible. On the one hand, the absence of derived-from cycles can remain a legality constraint, but, therefore, the merge operation should be such that it coalesces all artifacts (and related process) in a loop and removes all self-loops to ensure legality is preserved. On the other hand, the constraint on derived-from cycles can be lifted, but the OPM edges decorated with time information and involved in loops should have their time information updated, to reflect the constraints of Proposition 5.33.

5.5 Refinement

The OPM reference specification [MCF$^+$11] introduces the notion of *refinement* as a relation between two graphs: this relation expresses that one graph represents a more complete description of execution than another graph. The term refinement is inspired by the concept of specification refinement in formal methods [WD96]. The concept was only intuitively defined as follows: a graph is a refinement of another if dependencies that can be inferred in the original graph are "preserved" in the refinement. The purpose of this section is to formally ground such a notion of refinement in the context of our temporal semantics (Note A.4.10).

We fix two legal OPM graphs G and H for use in this section. We also define the following convenient notion.

Definition 5.34. The *logical closure* of a set of inequalities Σ, denoted by $\overline{\Sigma}$, is the set of logical consequences of Σ, i.e., $\overline{\Sigma} = \{\varphi \mid \Sigma \models \varphi\}$.

When context allows, we abbreviate logical closure to closure.

First, we define restriction of an arbitrary set of inequalities Σ to a subset of variables occurring in Σ.

Definition 5.35. Let Σ be a set of inequalities over a set of variables V. Let W be a subset of V. The *restriction* of Σ to W, denoted by $\Sigma|_W$, is the set

$$\Sigma|_W = \{u \preceq v \in \Sigma \mid u, v \in W\} .$$

Given our temporal semantics, the intuition of a refinement is the following. Graph H is a refinement of G if all the temporal constraints that can be inferred from $\mathrm{Th}(G)$ can also be inferred from $\mathrm{Th}(H)$. However, such definition is too broad, since refinements can replace nodes by others (say, when a process is implemented by composing two other processes); some temporal constraints of $\mathrm{Th}(G)$ may range over temporal variables that do not exist in $\mathrm{Th}(H)$. Hence, H is a refinement of G, if the temporal constraints that can be inferred from $\mathrm{Th}(G)$ over the common set of variables between H and G, can also be inferred from $\mathrm{Th}(H)$. Formally, the definition is expressed as follows.

Definition 5.36 (Refinement). H is a *refinement* of G if

$$\overline{\mathrm{Th}(G)}|_{\mathit{Vars}(G) \cap \mathit{Vars}(H)} \subseteq \overline{\mathrm{Th}(H)} .$$

Note that Theorem 5.14 can be effectively used to decide whether a given graph H is a refinement of a given graph G. Still, the definition of refinement is strictly semantic and does not provide much guidance towards constructing a refinement. An interesting open problem is to find a finite set of graph operations that all result in refinements, and such that every refinement can be obtained by using these operations.

5.5.1 Graph operations and refinement

In this section we investigate the relations between the graph operations defined in Section 5.4 and refinement. We first investigate the subgraph operation, the union and the intersection. Let G and H be two legal OPM graphs.

Proposition 5.37. *If H is a legal subgraph of G, then G is a refinement of H.*

Proof. Since H is a subgraph of G, it is clear from Definition 5.5 that $\mathrm{Th}(H) \subseteq \mathrm{Th}(G)$. Hence $\overline{\mathrm{Th}(H)}|_{Vars(H)\cap Vars(G)} = \overline{\mathrm{Th}(H)} \subseteq \overline{\mathrm{Th}(G)}$ as desired. □

Note that a legal subgraph of G is not necessarily a refinement of G. For instance, let G be a use–generate–derive triangle (A, B, P, r), and let H be a subgraph of G consisting of $P \xrightarrow{r} B$ and $A \xrightarrow{!} P$. Then H is legal, but is not a refinement of G: $\mathrm{create}(B) \preceq \mathrm{create}(A) \in \overline{\mathrm{Th}(G)}|_{Vars(G)\cap Vars(H)}$ yet $\mathrm{create}(B) \preceq \mathrm{create}(A) \notin \overline{\mathrm{Th}(H)}$.

The above proposition also applies to the union and intersection, and their operands.

Corollary 5.38. *If $G \cup H$ is legal, then $G \cup H$ is a refinement of G.*

Note that G is not necessarily a refinement of a legal $G \cup H$. For example, let G consist of $A \to B$ and node P and let H consist of $A \xrightarrow{!} P$ and node B. So $G \cup H$ is legal and consists of $A \to B$ and $A \xrightarrow{!} P$. Then G is not a refinement of $G \cup H$: $G \cup H \vdash P \to B$, so $\mathrm{create}(B) \preceq \mathrm{end}(P) \in \overline{\mathrm{Th}(G \cup H)}|_{Vars(G\cup H)\cap Vars(G)}$ yet $\mathrm{create}(B) \preceq \mathrm{end}(P) \notin \overline{\mathrm{Th}(G)}$.

However, if G and H are node-disjoint, then G is a refinement of $G \cup H$.

Corollary 5.39. *G is a refinement of $G \cap H$.*

Note that $G \cap H$ is not necessarily a refinement of G. For example, let G consists of $A \to Q$, $A \xrightarrow{!} P$ and $P \to Q$, and let H consist of $A \to Q$ and $P \to Q$. Then $G \cap H$ equals H, which is not a refinement of G: $\mathrm{create}(A) \preceq \mathrm{end}(P) \in \overline{\mathrm{Th}(G)}|_{Vars(G)\cap Vars(H)}$, yet $\mathrm{create}(A) \preceq \mathrm{end}(P) \notin \overline{\mathrm{Th}(G \cap H)}$.

Next we investigate the merge-renaming operation. Let H be a renaming of G by ρ_{Art}, ρ_{Proc} and ρ_{Roles} (conforming to Definition 5.27). Then G is not necessarily a refinement of H, or vice versa. For instance, let G consist of $A \to B$, let $\rho_{Art}(A) = B$ and $\rho_{Art}(B) = A$, then H consists of $B \to A$. They are clearly not each others refinements.

Still, if one needs to apply a merge-renaming on a graph before performing a union or an intersection, a merge-renaming operation that preserves the temporal constraints of the original would be advisable. This motivates the following restricted version of the merge-renaming operation:

Definition 5.40 (Proper merge-renaming). Let ρ_{Art}, ρ_{Proc} and ρ_{Roles} be as in Definition 5.30, and let ρ be the point-wise union of ρ_{Art}, ρ_{Proc} and ρ_{Roles}. Then we use $\rho(G)$ to denote the merge-renaming of G by ρ_{Art}, ρ_{Proc} and ρ_{Roles}.

We call a merge-renaming $\rho(G)$ *proper* if for any node (or role) x in G, if $\rho(x) \neq x$ but $\rho(x)$ is also a node (or role) in G, then $\rho(\rho(x)) = \rho(x)$.

Intuitively, in a proper merge-renaming operation, some nodes/roles are preserved, whereas all the others are either renamed into new ones or coalesced into preserved ones. Such an operation disallows arbitrary renaming and permutation of nodes/roles. Although it seems overly restrictive, it makes sense in the OPM model, because nodes and roles are actually *identifiers*. Hence, coalescing of nodes/roles, or renaming them into new ones, is a form of identity resolution. For example, witnesses of a car accident may speak of a "blue car" or a "Toyota". Later on, one realises the witnesses talked about the same car, so both "blue car" and "Toyota" should be renamed to the car's registration number (while "blue car" and "Toyota" become its annotations). Likewise, hospitals frequently admit unconscious patients under a temporary ID. Later on, they are either matched to an already known patient or to a new ID if the patient is admitted for the first time.

We conclude this section by showing that a proper and legal merge-renaming of a graph is indeed a refinement of the original graph.

Theorem 5.41. *Let G be a legal OPM graph, and let $\rho(G)$ be a proper and legal merge-renaming of G, for some ρ. Then $\rho(G)$ is a refinement of G.*

To prove the theorem we use the following auxiliary lemmas.

Lemma 5.42. *Let G be a legal OPM graph, and let $\rho(G)$ be a legal merge-renaming of G, for some ρ. Then if $G \vdash X \dashrightarrow Y$, also $\rho(G) \vdash \rho(X) \dashrightarrow \rho(Y)$, i.e., a legal merge-renaming preserves edge-inference.*

The above lemma is readily verified.

Lemma 5.43. *Let G be a legal OPM graph, and let $\rho(G)$ be a proper merge-renaming of G, for some ρ. If a node (or role) x belongs to both G and to the image of ρ, then $\rho(x) = x$.*

Proof. Indeed, if x is in the image of ρ, then there exist a node (role) y in G, such that $\rho(x) = y$. If $x = y$, then $\rho(x) = x$ holds immediately. If $x \neq y$, then $\rho(y) \neq y$, so $\rho(\rho(y)) = \rho(y)$ (by Definition 5.40). Thus $\rho(x) = x$ holds as desired. $\square$

Proof of Theorem 5.41. Let *Commons* $= \mathit{Vars}(G) \cap \mathit{Vars}(\rho(G))$. We need to prove the following:

$$\overline{\mathrm{Th}(G)}|_{\mathit{Commons}} \subseteq \overline{\mathrm{Th}(\rho(G))}\,.$$

For each inequality $\varphi \in \overline{\mathrm{Th}(G)}|_{\mathit{Commons}}$, we must show that φ belongs to $\overline{\mathrm{Th}(\rho(G))}$. We know from Theorem 5.14, illustrated in Figure 5.7, that each such inequality is associated with a pattern in G. Therefore, we need to find a similar pattern in $\rho(G)$ that produces *exactly* the same inequality.

Let $\varphi \in \overline{\mathrm{Th}(G)}|_{\mathit{Commons}}$. We follow the axioms and rules presented in Figure 5.7, the axioms first, to cover all possible forms that φ may assume:

(a) φ is $\mathrm{begin}(P) \preceq \mathrm{end}(P)$, for some P in G. Since $\mathrm{begin}(P), \mathrm{end}(P) \in$ *Commons*, P is also present in $\rho(G)$. By Axiom 1, $\mathrm{begin}(P) \preceq \mathrm{end}(P) \in \overline{\mathrm{Th}(\rho(G))}$.

(b) φ is $\mathrm{begin}(P) \preceq \mathrm{create}(A)$ or $\mathrm{create}(A) \preceq \mathrm{end}(P)$, for some $A \xrightarrow{!} P$ in G. Since $\mathrm{create}(A), \mathrm{begin}(P), \mathrm{end}(P) \in$ *Commons*, we also have A and P in $\rho(G)$. From $A \xrightarrow{!} P$ in G, by Definition 5.40, we have $\rho(A) \xrightarrow{!} \rho(P)$ in $\rho(G)$. We can apply Lemma 5.43 to A and P, obtaining $\rho(A) = A$ and $\rho(P) = P$. Hence, $A \xrightarrow{!} P$ is also in $\rho(G)$ and, by Axiom 2, both $\mathrm{begin}(P) \preceq \mathrm{create}(A)$ and $\mathrm{create}(A) \preceq \mathrm{end}(P)$ belong to $\overline{\mathrm{Th}(\rho(G))}$. Note that $A \xrightarrow{!} P$ in G and $A \xrightarrow{!} P$ in $\rho(G)$ need not have the same role.

(c) φ is one of the following: $\mathrm{begin}(P) \preceq \mathrm{use}(P, r, A)$, $\mathrm{use}(P, r, A) \preceq \mathrm{end}(P)$, or $\mathrm{create}(A) \preceq \mathrm{use}(P, r, A)$, for some $P \xrightarrow{r} A$ in G. The variable $\mathrm{use}(P, r, A)$ belongs to *Commons*; that is only possible if edge $P \xrightarrow{r} A$ is also present in $\rho(G)$. By Axiom 3, $\mathrm{begin}(P) \preceq \mathrm{use}(P, r, A)$, $\mathrm{use}(P, r, A) \preceq \mathrm{end}(P)$, and $\mathrm{create}(A) \preceq \mathrm{use}(P, r, A)$ belong to $\overline{\mathrm{Th}(\rho(G))}$.

(d) φ is $\mathrm{use}(P, r, B) \preceq \mathrm{create}(A)$, for some $G \bigtriangleup (A, B, P, r)$. For $P \xrightarrow{r} B$ in G we apply case c, so $P \xrightarrow{r} B$ is also in $\rho(G)$. For $A \xrightarrow{!} P$ in G, by case b, $A \xrightarrow{!} P$ belongs to $\rho(G)$. We can apply Lemma 5.43 to A, B, and r, obtaining $\rho(A) = A$, $\rho(B) = B$, and $\rho(r) = r$. For $A \xrightarrow{r} B$ in G, we apply Definition 5.40, so $\rho(A) \xrightarrow{\rho(r)} \rho(B)$ in $\rho(G)$, and thus $A \xrightarrow{r} B$ in $\rho(G)$.

Clearly, $\rho(G) \bigtriangleup (A, B, P, r)$, hence $\text{use}(P, r, B) \preceq \text{create}(A) \in \overline{\text{Th}(\rho(G))}$ (Axiom 8).

(e) φ is $\text{create}(B) \preceq \text{create}(A)$, for $G \vdash A \dashrightarrow B$. Since $\text{create}(A), \text{create}(B) \in$ *Commons*, A and B also belong to $\rho(G)$. From $G \vdash A \dashrightarrow B$, by Lemma 5.42, we have $\rho(G) \vdash \rho(A) \dashrightarrow \rho(B)$. We can apply Lemma 5.43 to A and B, obtaining $\rho(A) = A$ and $\rho(B) = B$. Hence $\rho(G) \vdash A \dashrightarrow B$ as desired and, by Rule 1, $\text{create}(B) \preceq \text{create}(A) \in \overline{\text{Th}(\rho(G))}$.

(f) φ is $\text{begin}(P) \preceq \text{create}(A)$, for $G \vdash A \dashrightarrow P$. Since $\text{create}(A)$, $\text{begin}(P)$, $\text{end}(P) \in$ *Commons*, A and P belong to $\rho(G)$. From $G \vdash A \dashrightarrow P$, by Lemma 5.42, $\rho(G) \vdash \rho(A) \dashrightarrow \rho(P)$. We can apply Lemma 5.43 to A and P, obtaining $\rho(A) = A$ and $\rho(P) = P$. Thus $\rho(G) \vdash A \dashrightarrow P$ and, by Rule 2, $\text{begin}(P) \preceq \text{create}(A) \in \overline{\text{Th}(\rho(G))}$.

(g) φ is $\text{create}(A) \preceq \text{end}(P)$, for $G \vdash P \dashrightarrow A$. Since $\text{create}(A)$, $\text{begin}(P)$, $\text{end}(P) \in$ *Commons*, A and P belong to $\rho(G)$. From $G \vdash P \dashrightarrow A$, by Lemma 5.42, we have $\rho(G) \vdash \rho(P) \dashrightarrow \rho(A)$. We can apply Lemma 5.43 to A and P, obtaining $\rho(A) = A$ and $\rho(P) = P$. Therefore, $\rho(G) \vdash P \dashrightarrow A$ and, by Rule 3, $\text{create}(A) \preceq \text{end}(P) \in \overline{\text{Th}(\rho(G))}$.

(h) φ is $\text{begin}(Q) \preceq \text{end}(P)$, for $G \vdash P \dashrightarrow Q$. Since $\text{begin}(P)$, $\text{end}(P)$, $\text{begin}(Q)$, $\text{end}(Q) \in$ *Commons*, P and Q belong to $\rho(G)$. From $G \vdash P \dashrightarrow Q$, by Lemma 5.42, $\rho(G) \vdash \rho(P) \dashrightarrow \rho(Q)$. We can apply Lemma 5.43 to P and Q, obtaining $\rho(P) = P$ and $\rho(Q) = Q$. Hence $\rho(G) \vdash P \dashrightarrow Q$ and, by Rule 4, $\text{begin}(Q) \preceq \text{end}(P) \in \overline{\text{Th}(\rho(G))}$.

(i) φ is $\text{create}(B) \preceq \text{use}(P, r, A)$, for $P \xrightarrow{r} A$ in G and $G \vdash A \dashrightarrow B$. By applying cases c and e to $P \xrightarrow{r} A$ in G and $G \vdash A \dashrightarrow B$, respectively, we have $P \xrightarrow{r} A$ in $\rho(G)$ and $\rho(G) \vdash A \dashrightarrow B$. By Rule 5, $\text{create}(B) \preceq \text{use}(P, r, A) \in \overline{\text{Th}(\rho(G))}$.

(j) φ is $\text{begin}(Q) \preceq \text{use}(P, r, A)$, for $P \xrightarrow{r} A$ in G and $G \vdash A \dashrightarrow Q$. By applying cases c and f to $P \xrightarrow{r} A$ in G and $G \vdash A \dashrightarrow Q$, respectively, we have $P \xrightarrow{r} A$ in $\rho(G)$ and $\rho(G) \vdash A \dashrightarrow Q$. By Rule 6, $\text{begin}(Q) \preceq \text{use}(P, r, A) \in \overline{\text{Th}(\rho(G))}$.

(k) φ is $\text{use}(P, r, C) \preceq \text{create}(A)$, for $G \bigtriangleup (B, C, P, r)$ and $G \vdash A \dashrightarrow B$. By applying cases d and e to $G \bigtriangleup (B, C, P, r)$ and $G \vdash A \dashrightarrow B$, respectively, we have $\rho(G) \bigtriangleup (B, C, P, r)$ and $\rho(G) \vdash A \dashrightarrow B$. By Rule 7, $\text{use}(P, r, C) \preceq \text{create}(A) \in \overline{\text{Th}(\rho(G))}$.

(l) φ is $\text{use}(P, r, B) \preceq \text{end}(Q)$, for $G \vartriangle (A, B, P, r)$ and $G \vdash Q \dashrightarrow A$. By applying cases d and g to $G \vartriangle (A, B, P, r)$ and $G \vdash Q \dashrightarrow A$, respectively, we have $\rho(G) \vartriangle \underline{(A, B, P, r)}$ and $\rho(G) \vdash Q \dashrightarrow A$. By Rule 8, $\text{use}(P, r, B) \preceq \text{end}(Q) \in \overline{\text{Th}(\rho(G))}$.

(m) φ is $\text{use}(P, r, B) \preceq \text{use}(Q, s, A)$, for $G \vartriangle (C, B, P, r)$ and $Q \xrightarrow{s} A$ in G, and either $A = C$ or $G \vdash A \dashrightarrow C$. By applying cases d and c to $G \vartriangle (C, B, P, r)$ and $Q \xrightarrow{s} A$, respectively, we obtain $\rho(G) \vartriangle (C, B, P, r)$ and $Q \xrightarrow{s} A$ in $\rho(G)$. If $A = C$, then A clearly belongs to G and to the image of ρ, so, by Lemma 5.43, $\rho(A) = A$. If $G \vdash A \dashrightarrow C$, then, by case e, $\rho(G) \vdash A \dashrightarrow C$. We can thus apply either Rule 9a or Rule 9b, so $\text{use}(P, r, B) \preceq \text{use}(Q, s, A) \in \overline{\text{Th}(\rho(G))}$.

5.6 Mapping a run of an NRC dataflow to an OPM graph

An obvious connection between the two parts of this dissertation would be a formal translation of runs of NRC dataflows to OPM graphs. Indeed, elsewhere [KV08] we have demonstrated such a translation. A complete reworking of this translation in the context of our formal model of a dataflow repository, remains as future work. This includes extending OPM with a profile for representing a specific kind of refinement between accounts of an OPM graph, suitable for modelling executions of subdataflows.

In Figure 5.11 we illustrate how a run can be translated into an OPM graph, by showing the OPM graph for the run from Example 3.2 (p. 42). The unique identities for the nodes are easily constructed from run-triples. For a triple $t = (\Phi \cdot [e], \sigma, v)$ we use $[t]$ as the identifier of the process, and label the process with e. We use $[t, v]$ as the identifier of the artifact representing v, and label it by v. For the artifact representing the value assignment, we simply use $[\sigma]$ as its identifier and label. This artifact is then used by all processes having σ in their run-triple. The generated-by edge with role *val* is from $[t, v]$ to $[t]$. The used-edge with role *env* is from $[t]$ to $[\sigma]$. Each $[t]$ also has used-edges from $[t]$ to the artifacts generated with role *val* by processes that represent the constituent subexpressions of e. We also need to explicitly assert which output was produced from which input, through the use-generate-derive triangles. In order to do so, we need roles for the used-edges and the corresponding derived-from edges. If e is not a unary expression, we take the same natural number or label, that is used in extending the subexpression path for its constituent subexpressions. If e is a unary expression, we simply take number 1.

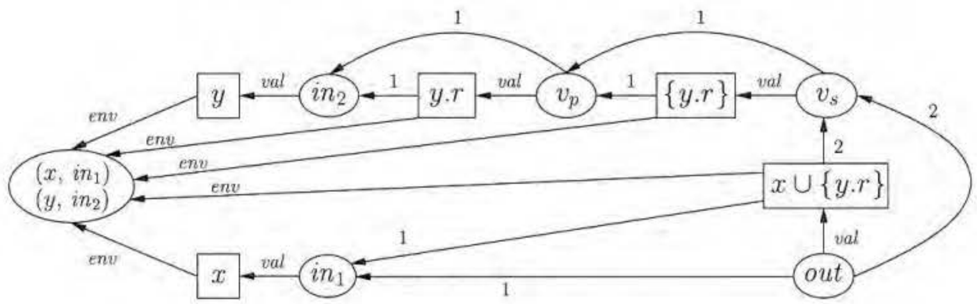


Figure 5.11: OPM graph for run from Example 3.2

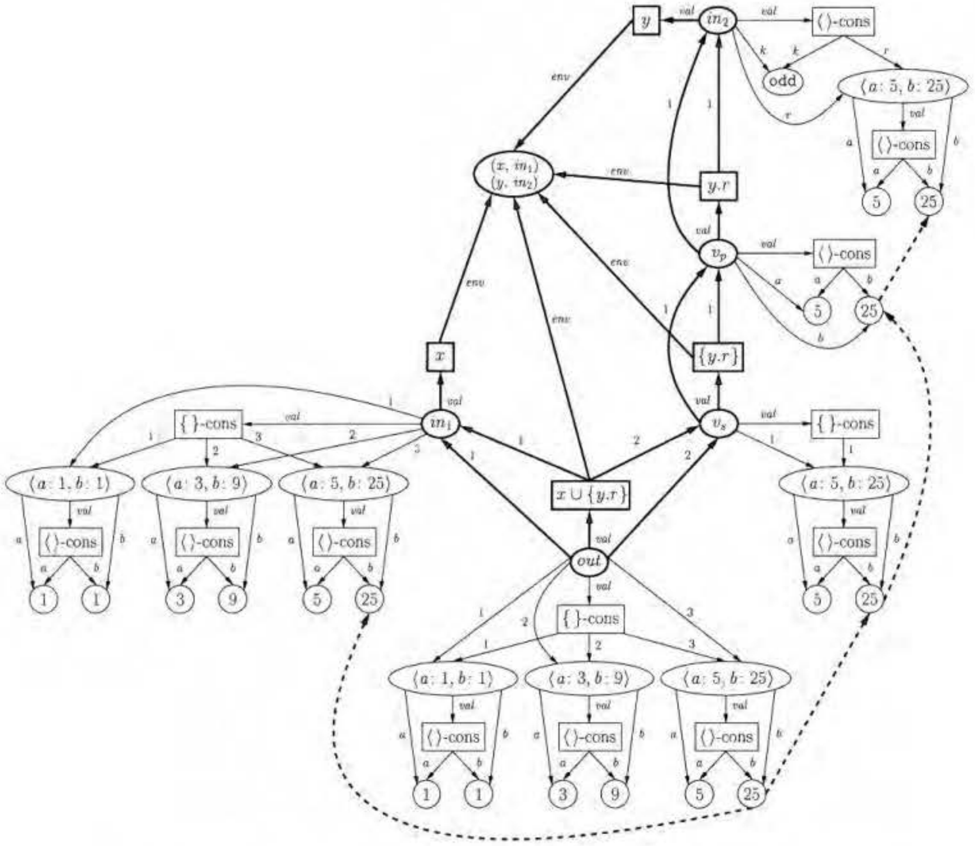


Figure 5.12: Using derived-from edges for subvalue provenance

As all artifacts are atomic, the derived-from edges only show dependencies between whole values. However, a complex value can also be translated into an OPM graph. Indeed, a tuple can be represented as a process that constructs a tuple from values, with the roles on the used-edges specifying the component. Similarly, a set can be represented as a set-constructing process, consuming values and producing a set containing these values. We can construct unique identifiers for all artifact nodes by extending $[t, v]$ with the corresponding subvalue path. For the process nodes, we can extend $[t, v]$ with a number in document order. As roles we can use labels for tuples, and natural numbers for sets. These OPM graphs for complex values can be incorporated into the graph for the run, however with a different account (an artifact may be produced by only one process for the account to be legal). Then we can use the rules for subvalue provenance to produce derived-from edges between subvalues (dashed bold edges), as we illustrate in Figure 5.12, for the subvalue provenance computation from Example 4.6 (p. 85 and Figure 4.1).

6 Outlook

In Chapter 3, we have presented a simple formal model of a dataflow repository, and a proof-of-concept representation in SQL/XML. In the context of queries used in the Provenance Challenges*, it would be interesting to (i) extend the model with necessary annotations for these types of queries, and (ii) research how we can tune the SQL/XML schema to ease the formulation of frequently posed queries. Moreover, it would be interesting to research what types of queries occur in a specific domain, for instance, in bioinformatics dataflows, and to design a special-purpose query language for this domain. Such a language could be implemented by compiling it into SQL/XML queries.

In Chapter 5 we have presented a temporal semantics for the Open Provenance Model. It would be interesting to adapt the temporal model to allow for duration in creating and using artifacts. Also, the notion of refinement in OPM is very general, and deals only with two OPM graphs at a time. It would be interesting to add a more specific concept of refinement, suitable for modelling the hierarchy of runs of nested subdataflows, each run represented by an OPM graph.

In Chapter 4, we have formulated inference rules for computing subvalue provenance. Although subvalue provenance can be expressed in OPM (with some representation for complex objects) by explicit derived-from edges, the advantage of using rules is that such edges can be added to an OPM graph afterwards, on demand. It would be interesting to identify frequently used data structures and data transformation operators in other workflow management systems, and to define similar subvalue provenance rules.

*`http://twiki.ipaw.info/bin/view/Challenge/WebHome`

Finally, it would be interesting to design a GUI allowing the user to explore an OPM graph at different levels. The user could start with a higher-level view of the execution, corresponding to the run of the dataflow without any runs of subdataflows, and only atomic representation of complex-objects. The GUI should then facilitate tracking subvalue provenance in the graph, providing subgraphs detailing complex values, or runs of subdataflows, on demand. Note that this type of view is determined by the way a dataflow was designed and executed. It could then serve as a starting point for defining other, higher-level views, like the user views defined by Davidson et al. [DCSC09, BCD08].

Bibliography

[ABAL10] Manish Anand, Shawn Bowers, Ilkay Altintas, and Bertram Ludäscher. Approaches for exploring and querying scientific workflow provenance graphs. In Deborah McGuinness, James Michaelis, and Luc Moreau, editors, *Provenance and Annotation of Data and Processes*, volume 6378 of *Lecture Notes in Computer Science*, pages 17–26. Springer Berlin / Heidelberg, 2010.

[AIL98] Anastassia Ailamaki, Yannis E. Ioannidis, and Miron Livny. Scientific workflow management by database management. In Maurizio Rafanelli and Matthias Jarke, editors, *SSDBM*, pages 190–199. IEEE Computer Society, 1998.

[BCD08] Olivier Biton, Sarah Cohen-Boulakia, and Susan B. Davidson. Querying and managing provenance through user views in scientific workflows. In *Proceedings 24th International Conference on Data Engineering*, pages 1072–1081. IEEE, 2008.

[BCV08] Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions on Database Systems (TODS)*, 33:28:1–28:47, December 2008.

[BD08] Roger S. Barga and Luciano A. Digiampietri. Automatic capture and efficient storage of e-Science experiment provenance. *Concurrency and Computation: Practice and Experience*, 20(5):419–429, 2008.

[BEKM08] Catriel Beeri, Anat Eyal, Simon Kamenkovich, and Tova Milo. Querying business processes with BP-QL. *Information Systems*, 33(6):477–507, 2008.

[BEMP07] Catriel Beeri, Anat Eyal, Tova Milo, and Alon Pilberg. Monitoring business processes with queries. In *Proceedings 33rd In-*

ternational Conference on Very Large Databases, pages 603–614. ACM, 2007.

[BF05] Rajendra Bose and James Frew. Lineage retrieval for scientific data processing: A survey. *ACM Computing Surveys*, 37(1):1–28, March 2005.

[BKT01] Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and Where: A characterization of data provenance. In Jan Van den Bussche and Victor Vianu, editors, *ICDT*, volume 1973 of *Lecture Notes in Computer Science*, pages 316–330. Springer-Verlag, 2001.

[Blo06] Hendrik Blockeel. Experiment databases: A novel methodology for experimental research. In Francesco Bonchi and Jean-Franois Boulicaut, editors, *Knowledge Discovery in Inductive Databases*, volume 3933 of *Lecture Notes in Computer Science*, pages 72–85. Springer Berlin / Heidelberg, 2006.

[BML08] Shawn Bowers, Timothy M. McPhillips, and Bertram Ludäscher. Provenance in collection-oriented scientific workflows. *Concurrency and Computation: Practice and Experience*, 20:519–529, 2008.

[BNTW95] Peter Buneman, Shamim Naqvi, Val Tannen, and Limsoon Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149:3–48, 1995.

[Bro08] Allen Brown. Enforcing the scientific method. In Juliana Freire, David Koop, and Luc Moreau, editors, *Provenance and Annotation of Data and Processes*, volume 5272 of *Lecture Notes in Computer Science*, pages 2–2. Springer Berlin / Heidelberg, 2008.

[BV07] Hendrik Blockeel and Joaquin Vanschoren. Experiment databases: Towards an improved experimental methodology in machine learning. In Joost Kok, Jacek Koronacki, Ramon Lopez de Mantaras, Stan Matwin, Dunja Mladenic, and Andrzej Skowron, editors, *Knowledge Discovery in Databases: PKDD 2007*, volume 4702 of *Lecture Notes in Computer Science*, pages 6–17. Springer Berlin / Heidelberg, 2007.

[CAA08] James Cheney, Umut A. Acar, and Amal Ahmed. Provenance traces. *The Computing Research Repository*, abs/0812.0564, 2008.

[CCT09] James Cheney, Laura Chiticarius, and Wang-Chiew Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[CCW03] Jing Chen, Su-Yun Chung, and Limsoon Wong. The Kleisli query system as a backbone for bioinformatics data integration and analysis. In Zoé Lacroix and Terence Critchlow, editors, *Bioinformatics: Managing Scientific Data*, chapter 6, pages 147–187. Morgan Kaufmann, 2003.

[CFV$^+$08] Ben Clifford, Ian T. Foster, Jens-S. Voeckler, Michael Wilde, and Yong Zhao. Tracking provenance in a virtual data grid. *Concurrency and Computation: Practice and Experience*, 20:565–575, 2008.

[Che10] James Cheney. Causality and the semantics of provenance. In S. Barry Cooper, Prakash Panangaden, and Elham Kashefi, editors, *DCM*, volume 26 of *EPTCS*, pages 63–74, 2010.

[CJ10] Adriane Chapman and H. V. Jagadish. Understanding provenance black boxes. *Distributed and Parallel Databases*, 27:139–167, 2010.

[CM95] I.-Min A. Chen and Victor M. Markowitz. An overview of the object protocol model (OPM) and the OPM data management tools. *Information Systems*, 20(5):393–418, 1995.

[CT09] Sarah Cohen-Boulakia and Wang-Chiew Tan. Provenance in scientific databases. In LING LIU and M. TAMER ZSU, editors, *Encyclopedia of Database Systems*, pages 2202–2207. Springer US, 2009.

[DCM] DCMI. The Dublin Core (R) Metadata Initiative home page. `http://dublincore.org/`.

[DCSC09] Susan B. Davidson, Yi Chen, Peng Sun, and Sarah Cohen-Boulakia. On user views in scientific workflow systems. In Juliana Freire, Paolo Missier, and Satya Sanket Sahoo, editors, *SWPM*, volume 526 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2009.

[DF08] Susan B. Davidson and Juliana Freire. Provenance and scientific workflows: Challenges and opportunities, 2008. `http://www.vistrails.org/index.php?title=Tutorials/SIGMOD2008`.

[DW04] Susan B. Davidson and Limsoon Wong. The Kleisli approach to data transformation and integraion. In P.M.D. Gray, L. Kerschberg, P.J.H. King, and A. Poulovassilis, editors, *The Functional Approach to Data Management*, chapter 6, pages 135–165. Springer, 2004.

[EM04] Andrew Eisenberg and Jim Melton. Advancements in SQL/XML. *SIGMOD Record*, 33:79–86, 2004.

[FK04] Ian T. Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Elsevier, 2nd edition, 2004.

[FKSS08] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. Provenance for computational tasks: A survey. *Computing in Science and Engineering*, 10:11–21, 2008.

[FMS08] James Frew, Dominic Metzger, and Peter Slaughter. Automatic capture and reconstruction of computational provenance. *Concurrency and Computation: Practice and Experience*, 20(5):485–496, 2008.

[FSC$^+$06] Juliana Freire, Cláudio T. Silva, Steven P. Callahan, Emanuele Santos, Carlos Eduardo Scheidegger, and Huy T. Vo. Managing rapidly-evolving scientific workflows. In Luc Moreau and Ian T. Foster, editors, *IPAW*, volume 4145 of *Lecture Notes in Computer Science*, pages 10–18. Springer, 2006. keynote.

[FVWZ02] Ian T. Foster, Jens-S. Vöckler, Michael Wilde, and Yong Zhao. Chimera: A virtual data system for representing, querying and automating data derivation. In *Proceedings of the 14th International Conference on Scientific and Statistical Database Management*, SSDBM 2002, pages 37–46, Washington, DC, USA, 2002. IEEE Computer Society.

[GCM$^+$11] Luiz M. R. Gadelha Jr., Ben Clifford, Marta Mattoso, Michael Wilde, and Ian T. Foster. Provenance management in Swift. *Future Generation Computer Systems*, 27(6):775–780, 2011.

[GDE$^+$07] Yolanda Gil, Ewa Deelman, Mark Ellisman, Thomas Fahringer, Geoffrey Fox, Dennis Gannon, Carole A. Goble, Miron Livny, Luc Moreau, and Jim Myers. Examining the challenges of scientific workflows. *IEEE Computer*, 40(12):26–34, 2007.

[GJZ06] Sumedha Gunewardena, Peter Jeavons, and Zhang Zhaolei. Enhancing the prediction of transcription factor binding sites by incorporating structural properties and nucleotide covariations. *Journal of Computational Biology*, 13(4):929–45, 2006.

[GV04] Kris Gevaert and Joël Vandekerckhove. COFRADIC(TM): the Hubble telescope of proteomics. *Drug Discovery Today: TARGETS*, 3(2, Supplement 1):16–22, 2004.

[HKS$^+$07] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. A formal model of dataflow repositories. In Sarah Cohen-Boulakia and Val Tannen, editors, *DILS*, volume 4544 of *Lecture Notes in Computer Science*, pages 105–121. Springer, 2007.

[HKS$^+$08] Jan Hidders, Natalia Kwasnikowska, Jacek Sroka, Jerzy Tyszkiewicz, and Jan Van den Bussche. DFL: A dataflow language based on Petri nets and nested relational calculus. *Information Systems*, 33(3):261–284, 2008.

[HP05] Joseph Y. Halpern and Judea Pearl. Causes and Explanations: A Structural-Model Approach. Part I: Causes. *British Journal for the Philosophy of Science*, 56(4):843–887, 2005.

[HSBMR08] David A. Holland, Margo I. Seltzer, Uri Braun, and Kiran-Kumar Muniswamy-Reddy. PASSing the provenance challenge. *Concurrency and Computation: Practice and Experience*, 20(5):531–540, 2008.

[KCL06] Natalia Kwasnikowska, Yi Chen, and Zoé Lacroix. Modeling and storing scientific protocols. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *OTM Workshops (1)*, volume 4277 of *Lecture Notes in Computer Science*, pages 730–739. Springer, 2006.

[KDG$^+$08] Jihie Kim, Ewa Deelman, Yolanda Gil, Gaurang Mehta, and Varun Ratnakar. Provenance trails in the Wings-Pegasus system. *Concurrency and Computation: Practice and Experience*, 20:587–597, 2008.

[KLL$^+$07] Michel Kinsy, Zoé Lacroix, Christophe Legendre, Piotr Włodarczyk, and Nadia Yacoubi. ProtocolDB: Storing scientific protocols with a domain ontology. In Mathias Weske, Mohand-Sad Hacid, and Claude Godart, editors, *Web Information Systems Engineering—WISE 2007 Workshops*, volume 4832 of *Lecture Notes in Computer Science*, pages 17–28. Springer Berlin / Heidelberg, 2007.

[KMV10] Natalia Kwasnikowska, Luc Moreau, and Jan Van den Bussche. A formal account of the Open Provenance Model. Technical Report ECS Technical Report 21819, University of Southampton, December 2010.

[KV08] Natalia Kwasnikowska and Jan Van den Bussche. Mapping the NRC dataflow model to the Open Provenance Model. In Juliana Freire, David Koop, and Luc Moreau, editors, *Provenance and Annotation of Data and Processes*, volume 5272 of *Lecture Notes in Computer Science*, pages 3–16. Springer-Verlag, Berlin, Heidelberg, 2008.

[LAB$^+$06] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger-Frank, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice And Experience*, 18(10):1039–1065, 2006. published online: 13 Dec 2005.

[Lam78] Leslie Lamport. Ti clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978.

[LKM$^+$09] Zoé Lacroix, Cartik R. Kothari, Peter Mork, Mark Wilkinson, and Sarah Cohen-Boulakia. Biological metadata management. In LING LIU and M. TAMER ZSU, editors, *Encyclopedia of Database Systems*, pages 215–219. Springer US, 2009.

[ŁKO$^+$06] Marta Łuksza, Bogusław Kluge, Jerzy Ostrowski, Jakub Karczmarski, and Anna Gambin. Efficient model-based clustering for LC-MS data. In Philipp Bücher and Bernard Moret, editors, *Algorithms in Bioinformatics*, volume 4175 of *Lecture Notes in Computer Science*, pages 32–43. Springer Berlin / Heidelberg, 2006.

[Mat89] Friedemann Mattern. Virtual time and global states of distributed systems. In M. Cosnard et al., editors, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Amsterdam, 1989. Elsevier Science Publishers.

[MBL06] Timothy McPhillips, Shawn Bowers, and Bertram Ludäscher. Collection-oriented scientific workflows for integrating and analyzing biological data. In Ulf Leser, Felix Naumann, and Barbara A. Eckman, editors, *Data Integration in the Life Sciences*, volume 4075 of *Lecture Notes in Computer Science*, pages 248–263. Springer, 2006.

[MBZ$^+$08] Paolo Missier, Khalid Belhajjame, Jun Zhao, Marco Roos, and Carole A. Goble. Data lineage model for Taverna workflows with lightweight annotation requirements. In Juliana Freire, David Koop, and Luc Moreau, editors, *Provenance and Annotation of Data and Processes*, volume 5272 of *Lecture Notes in Computer Science*, pages 17–30. Springer Berlin / Heidelberg, 2008.

[MCF$^+$11] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul T. Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Jan Van den Bussche. The Open Provenance Model core specification (v1.1). *Future Generation Computer Systems*, 27(6):743–756, 2011.

[MDF$^+$10] Luc Moreau (editor), Li Ding, Joe Futrelle, Daniel Garijo Verdejo, Paul T. Groth, Mike Jewell, Simon Miles, Paolo Missier, Jeff Pan, and Jun Zhao. Open Provenance Model (OPM) OWL Specification. Technical report, openprovenance.org, 2010. `http://openprovenance.org/model/opmo`.

[MF08] Robert E. McGrath and Joe Futrelle. Reasoning about provenance with OWL and SWRL rules. In *AAAI Spring Symposium 2008 "AI Meets Business Rules and Process Management"*, 2008.

[MG11] Paolo Missier and Carole A. Goble. Workflows to open provenance graphs, round-trip. *Future Generation Computer Systems*, 27(6):812–819, 2011.

[MGBM07] Simon Miles, Paul T. Groth, Miguel Branco, and Luc Moreau. The requirements of using provenance in e-Science experiments. *Journal of Grid Computing*, 5:1–25, 2007.

[MGM$^+$08] Simon Miles, Paul T. Groth, Steve Munroe, Sheng Jiang, Thibaut Assandri, and Luc Moreau. Extracting causal graphs from an open provenance data model. *Concurrency and Computation: Practice and Experience*, 20(5):577–586, 2008.

[MLA$^+$08] Luc Moreau, Bertram Ludäscher, Ilkay Altintas, Roger S. Barga, Shawn Bowers, Steven Callahan, George Chin, Jr., Ben Clifford, Shirley Cohen, Sarah Cohen-Boulakia, Susan B. Davidson, Ewa Deelman, Luciano Digiampietri, Ian T. Foster, Juliana Freire, James Frew, Joe Futrelle, Tara Gibson, Yolanda Gil, Carole A. Goble, Jennifer Golbeck, Paul T. Groth, David A. Holland, Sheng Jiang, Jihie Kim, David Koop, Ales Krenek, Timothy McPhillips, Gaurang Mehta, Simon Miles, Dominic Metzger, Steve Munroe, Jim Myers, Beth Plale, Norbert Podhorszki, Varun Ratnakar, Emanuele Santos, Carlos Scheidegger, Karen Schuchardt, Margo Seltzer, Yogesh L. Simmhan, Cláudio T. Silva, Peter Slaughter, Eric Stephan, Robert D. Stevens, Daniele Turi, Huy Vo, Mike Wilde, Jun Zhao, and Yong Zhao. Special Issue: The First Provenance Challenge. *Concurrency and Computation: Practice and Experience*, 20:409–418, 2008.

[Mor10] Luc Moreau. The foundations for provenance on the web. *Foundations and Trends in Web Science*, 2:99–241, 2010.

[NDK$^+$06] Jean-Paul Noben, Debora Dumont, Natalia Kwasnikowska, Peter Verhaert, Veerle Somers, Raymond Hupperts, Piet Stinissen, and Johan Robben. Lumbar cerebrospinal fluid proteome in multiple sclerosis: Characterization by ultrafiltration, liquid chromatography, and mass spectrometry. *Journal of Proteome Research*, 5(7):1647–1657, 2006.

[ÖCKM06] Fatma Özcan, Donald D. Chamberlin, Krishna G. Kulkarni, and Jan-Eike Michels. Integration of SQL and XQuery in IBM DB2. *IBM Systems Journal*, 45(2):245–270, 2006.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[SGBB01] Robert D. Stevens, Carole A. Goble, Patricia G. Baker, and Andy Brass. A classification of tasks in bioinformatics. *Bioinformatics*, 17(2):180–188, 2001.

[SGM11] Yogesh Simmhan, Paul T. Groth, and Luc Moreau. Special section: The third Provenance Challenge on using the Open Provenance Model for interoperability. *Future Generation Computer Systems*, 27(6):737–742, 2011.

[SH09a] Jacek Sroka and Jan Hidders. Towards a formal semantics for the process model of the Taverna workbench. part I. *Fundamenta Informaticae*, 92:279–299, August 2009.

[SH09b] Jacek Sroka and Jan Hidders. Towards a formal semantics for the process model of the Taverna workbench. part II. *Fundamenta Informaticae*, 92:373–396, December 2009.

[SHMG10] Jacek Sroka, Jan Hidders, Paolo Missier, and Carole A. Goble. A formal semantics for the Taverna 2 workflow model. *Journal of Computer and System Sciences*, 76(6):490–508, 2010. Special Issue: Scientific Workflow 2009, The 2nd International Workshop on Workflow Management and Application in Grid Environments & The 3rd International Workshop on Workflow Management and Applications in Grid Environments.

[SKDN05] Srinath Shankar, Ameet Kini, David J. DeWitt, and Jeffrey Naughton. Integrating databases and workflow systems. *SIGMOD Record*, 34(3):5–11, September 2005.

[SPG05] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-Science. *SIGMOD Record*, 34(3):31–36, 2005.

[SPG08] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. Query capabilities of the Karma provenance framework. *Concurrency and Computation: Practice and Experience*, 20(5):441–451, 2008.

[Tel94] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[TMG$^+$07] Daniele Turi, Paolo Missier, Carole A. Goble, David De Roure, and Tom Oinn. Taverna workflows: Syntax and semantics. *e-Science and Grid Computing, International Conference on*, 0:441–448, 2007.

[Ull90] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems: Vol. 2*. W. H. Freeman & Co., New York, NY, USA, 1990.

[vdAvH04] Wil van der Aalst and Kees van Hee. *Workflow Management*. MIT Press, 2004.

[VGV96] Jan Van den Bussche, Dirk Van Gucht, and Gottfried Vossen. Reflective programming in the relational algebra. *Journal of Computer and System Sciences*, 52(3):537–549, 1996.

[VVV04] Jan Van den Bussche, Stijn Vansummeren, and Gottfried Vossen. Meta-SQL: Towards practical meta-querying. In *Advances in Database Technology - EDBT 2004*, volume 2992 of *Lecture Notes in Computer Science*, pages 823–825. Springer, 2004. system demonstration.

[VVV05] Jan Van den Bussche, Stijn Vansummeren, and Gottfried Vossen. Towards practical meta-querying. *Information Systems*, 30(4):317–332, 2005.

[WD96] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall, Upper Saddle River, NJ, USA, 1996.

[ZGST08] Jun Zhao, Carole A. Goble, Robert D. Stevens, and Daniele Turi. Mining Taverna's semantic web of provenance. *Concurrency and Computation: Practice and Experience*, 20:463–472, 2008.

A

Notes and auxiliaries

A.1 Sequences

We often employ sequences over a set X, i.e., functions from a chosen index set I to X. We assume I to be totally ordered. Unless otherwise stated, we take an initial finite subset of $\mathbb{N}^*$ for I. We use $SEQ(X)$ to denote the set of all sequences over a set X.

When writing down sequences, we separate the elements by commas, and we use square brackets ("[" and "]") as delimiters. The empty sequence is denoted by "[]". The concatenation operator for sequences is denoted by "$\cdot$".

Let p, q and s be non-empty sequences over a set X. If $s = p \cdot q$, then we call p a *prefix* of s, and q a *suffix* of s.

Let Y_i, for $i \in \{1, \ldots, n\}$, be sets. Let $Q \subseteq SEQ(X) \times Y_1 \times \cdots \times Y_n$ and let $x_j \in X$, for $j \in \{1, \ldots, m\}$. We introduce the following notation for concatenating a prefix to the first component of each element of Q:

$$[x_1, \ldots, x_m] \cdot Q \overset{def}{=} \{([x_1, \ldots, x_m] \cdot s,\ y_1, \ldots, y_n) \mid (s,\ y_1, \ldots, y_n) \in Q\} \quad (A.1)$$

We call it *prefixing* of Q by $[x_1, \ldots, x_m]$.

If in a sequence $[x_1, \ldots, x_n]$ the order of its elements is of no consequence, we can write the sequence as $[x_i \mid i \in \{1, \ldots, n\}]$.

A.2 Constructing a wrapper relation

Let R be a partial relation from $I_1 \times \ldots \times I_n$ to O. Let $\tau_1 \times \ldots \times \tau_n$ and τ_{out} be complex types, such that

- $I_i \subseteq [\![\tau_i]\!]$, for $i \in \{1, \ldots, n\}$, and
- $O \subseteq [\![\tau_{out}]\!]$.

Let OK and *value* be labels, and let I be the greatest subset of $I_1 \times \ldots \times I_n$ over which R is total, i.e.,

$$I = \{(w_1, \ldots, w_n) \in I_1 \times \ldots \times I_n \mid \exists\, v \in O\colon (w_1, \ldots, w_n, v) \in R\}.$$

We construct a *wrapper relation* $R^{\Box}$ for R as a total relation from $[\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$ to $[\![\langle OK : \mathbf{Boolean}, value : \tau_{out}\rangle]\!]$ as follows:

(1) For each $(w_1, \ldots, w_n, v) \in R$ we add

$$(w_1, \ldots, w_n, \langle OK : \texttt{true}, value : v\rangle)$$

to $R^{\Box}$. We preserve thus the behaviour of the original relation R.

(2) For each $(w_1, \ldots, w_n) \in (([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \setminus I)$ we add

$$(w_1, \ldots, w_n, \langle OK : \texttt{false}, value : z\rangle)$$

to $R^{\Box}$, where z stands for an arbitrary element of $[\![\tau_{out}]\!]$. We ensure thus the totality of $R^{\Box}$.

For a value $\langle OK : b, value : v\rangle$, with $b \in [\![\mathbf{Boolean}]\!]$ and $v \in [\![\tau_{out}]\!]$, value v is a semantically correct value only if $b = \texttt{true}$.

A.3 DTDs

A.3.1 DTD for complex values

```
<!ELEMENT set (emptyset | base+ | set+ | tuple+) >
<!ATTLIST set
          valID ID #REQUIRED >
<!ELEMENT emptyset EMPTY >
<!ATTLIST emptyset
          valID ID #REQUIRED >
<!ELEMENT base (#PCDATA) >
<!ATTLIST base
          valID ID #REQUIRED
          kind (b | B) #REQUIRED >
<!ELEMENT tuple (lbl, (emptyset | base | set | tuple))+ >
<!ATTLIST tuple
          valID ID #REQUIRED >
<!ELEMENT lbl (#PCDATA) >
```

Attribute `valID` is used to indentify occurrences of subvalues. If the complex value is stored in table `Values`, we assume that the `valID` attribute of the root element of the value stored in `Values(value)` matches the corresponding value in `Values(vID)`. If the complex value is stored in tables `Sets` and `Tuples`, we assume that the `valID` attribute of tuple and set elements contains the corresponding value of `Sets(vID)` and `Tuples(vID)`, respectively. For element `base`, if attribute `kind` is `B`, then attribute `valID` must be a foreign key to `BaseValues(vID)`.

A.3.2 DTD for complex types

```
<!ELEMENT settype (bottom | basetype | settype | tupletype) >
<!ATTLIST settype
          typeID NMTOKEN #REQUIRED >
<!ELEMENT bottom  EMPTY >
<!ATTLIST bottom
          typeID (bottom) #FIXED "bottom" >
<!ELEMENT basetype EMPTY >
<!ATTLIST basetype
          typeID NMTOKEN #REQUIRED >
<!ELEMENT tupletype (lbl, (basetype | settype | tupletype))+ >
<!ATTLIST tupletype
          typeID NMTOKEN #REQUIRED >
<!ELEMENT lbl (#PCDATA) >
```

Attribute `typeID` contains the name of the complex type, which serves as its identifier in the dataflow repository, i.e., it is a foreign key to one of the following: `BaseTypes(tID)` (for elements `bottom` and `basetype`), or `SetTypes(tID)`, or `TupleTypes(tID)`.

A.3.3 DTD for type assignments

```
<!ELEMENT tassign (vartype)* >
<!ELEMENT vartype (var, (tID | type)) >
<!ELEMENT var (#PCDATA) >
<!ELEMENT tID (#PCDATA) >
<!ATTLIST tID
          kind (b | s | t) #REQUIRED >
<!ELEMENT type (basetype | settype | tupletype) >
```

Elements `basetype`, `settype`, and `tupletype` are already defined in Section A.3.2. Element `var` stands for a free variable of a dataflow. Element var is paired with either (i) element type, containing a complex type in XML format; or (ii) element tID, containing a complex type identifier, i.e., a foreign key to one of the following: BaseTypes(tID) (kind="b"), SetTypes(tID) (kind="s") or TupleTypes(tID) (kind="t").

A.3.4 DTD for signature assignments

```
<!ELEMENT sassign (service)* >
<!ELEMENT service (partype)* >
<!ATTLIST service
          name ID #REQUIRED >
<!ELEMENT partype (par, (tID | type)) >
<!ELEMENT par (#PCDATA) >
```

Elements tID and type are already defined in Section A.3.3. Attribute name contains the service name of a service-call expression. The definition of element partype is similar to that of vartype in Section A.3.3, except that element par stands here for a parameter name in the signature of the service name.

A.3.5 DTD for NRC expressions

```
<!ENTITY  % base "(const | var | emptyExpr)" >
<!ENTITY  % composite "(setExpr | union | flatten |
                       tupleExpr | project | for |
                       eqTest | emptyTest | if | let | call)" >
<!ENTITY  % subExpr "(%base; | %composite;)" >
<!ELEMENT expr %subExpr; >
<!ATTLIST expr
          eID ID #REQUIRED >
<!ELEMENT const (#PCDATA) >
<!ATTLIST const
          eID ID #REQUIRED >
<!ELEMENT var (#PCDATA) >
<!ATTLIST var
          eID ID #REQUIRED >
<!ELEMENT emptyExpr EMPTY >
<!ATTLIST emptyExpr
          eID ID #REQUIRED >
<!ELEMENT setExpr %subExpr; >
<!ATTLIST setExpr
          eID ID #REQUIRED >
<!ELEMENT union (%subExpr;, %subExpr;) >
<!ATTLIST union
```

```
          eID ID #REQUIRED >
<!ELEMENT flatten %subExpr; >
<!ATTLIST flatten
          eID ID #REQUIRED >
<!ELEMENT tupleExpr (lbl, %subExpr;)+>
<!ATTLIST tupleExpr
          eID ID #REQUIRED >
<!ELEMENT project (%subExpr;, lbl)>
<!ATTLIST project
          eID ID #REQUIRED >
<!ELEMENT for (var, %subExpr;, %subExpr;) >
<!ATTLIST for
          eID ID #REQUIRED >
<!ELEMENT eqTest (%subExpr;, %subExpr;) >
<!ATTLIST eqTest
          eID ID #REQUIRED >
<!ELEMENT emptyTest %subExpr; >
<!ATTLIST emptyTest
          eID ID #REQUIRED >
<!ELEMENT if (%subExpr;, %subExpr;, %subExpr;) >
<!ATTLIST if
          eID ID #REQUIRED >
<!ELEMENT let (var, %subExpr;, %subExpr;) >
<!ATTLIST let
          eID ID #REQUIRED >
<!ELEMENT call (service, %subExpr;, (%subExpr;)*) >
<!ATTLIST call
          eID ID #REQUIRED >
<!ELEMENT lbl (#PCDATA) >
<!ELEMENT service (#PCDATA) >
```

Attribute `eID` is used to indentify occurrences of subexpressions. We assume that the top element has the name of the dataflow as its `eID`, and others have an `eID` composed of `e` followed by a number in document order, starting at one. The top-level NRC expression must have `eID="e1"`.

A.3.6 DTD for value assignments

```
<!ELEMENT vassign (varval)* >
<!ELEMENT varval (var, (vID | value)) >
<!ELEMENT var (#PCDATA) >
<!ELEMENT vID (#PCDATA) >
<!ATTLIST vID
          kind (B | s | t) #REQUIRED >
<!ELEMENT value (emptyset | base | set | tuple) >
```

Elements `emptyset`, `base`, `set`, and `tuple` are already defined in Section A.3.1. Element `var` stands for a free variable of a dataflow. Element `var` is paired with either (i) element `value`, containing a complex value in XML format; or (ii) element `vID`, containing a complex value identifier, i.e., a foreign key to one of the following: `BaseValues(vID)` (kind="B"), `Sets(vID)` (kind="s") or `Tuples(vID)` (kind="t"). Base values not stored in table `BaseValues` must be in XML format.

A.3.7 DTD for binding trees

```
<!ELEMENT btree (extentry | subentry)* >
<!ATTLIST btree
          dID CDATA #REQUIRED >
<!ELEMENT extentry (service, ext, extpair*) >
<!ELEMENT extpair (epar, spar) >
<!ELEMENT subentry (service, sub, subpair*, btree) >
<!ELEMENT subpair (var, spar) >
<!ELEMENT service (#PCDATA) >
<!ELEMENT ext (#PCDATA) >
<!ELEMENT sub (#PCDATA) >
<!ELEMENT spar (#PCDATA) >
<!ELEMENT epar (#PCDATA) >
<!ELEMENT var (#PCDATA) >
```

Element `extentry` corresponds to a leaf binding a service name to an external service, while element `subentry` corresponds to a node binding a service name to a subdataflow. Element `service` stands for a service name, and element `spar` for a parameter name in the signature of the service name. Element `ext` is the identifier of an external service, and is a foreign key to `External(extID)`. Element `epar` stands for a parameter name in the signature of the external service, i.e., `ext` and `epar` are a foreign key to `ExtSigs(extID,par)`. Element `sub` is a subdataflow identifier, and is a foreign key to `Dataflows(dID)`. Element `var` stands for a free variable of the subdataflow, i.e., `sub` and `var` are a foreign key to `Variables(dID,var)`. Finally, attribute `dID` of `btree`, a grandchild `service` of `btree`, and a child `spar` of sibling `subpair` of `service`, are a foreign key to `Signatures(dID,service,par)`. (Sibling `btree` of an element `sub` must have the contents of `sub` in attribute `dID`.)

Table A.1: Table for Note A.4.4.

OPM reference specification	This work
`Used`	*Used*! and *Used*
`WasGeneratedBy`	*GeneratedBy*! and *GeneratedBy*
`WasDerivedFrom`	*DerivedFrom*! and *DerivedFrom*
`Used`* (asserted)	*Used*
`Used`* (inferred)	$\dashrightarrow$
`WasGeneratedBy`* (asserted)	*GeneratedBy*
`WasGeneratedBy`* (inferred)	$\dashrightarrow$
`WasDerivedFrom`* (asserted)	*DerivedFrom*
`WasDerivedFrom`* (inferred)	$\dashrightarrow$
`WasTriggeredBy` (asserted)	*InformedBy*
`WasTriggeredBy` (inferred)	$\dashrightarrow$
`WasControlledBy`	n/a

A.4 OPM-specific notes

Note A.4.1. The OPM reference specification also includes agents, which are entities controlling processes. We do not formalize such a concept here, since it is the subject of multiple ongoing discussions.

Note A.4.2. The OPM reference specification also includes edges of the type WasControlledBy (from a process to an agent) which are not included in this work, because we do not model agents. Furthermore, we prefer to adopt the term *informed-by* rather than *was-triggered-by* since its informal meaning is more aligned to its formal definition.

Note A.4.3. The OPM reference specification does not associate a role with a was-derived-from edge. It is a contribution of this work to have identified the need of this role for defining a temporal semantics of OPM.

Note A.4.4. Table A.1 maps sets in the OPM reference specification to sets or notations introduced in this work.

Note A.4.5. The OPM reference specification has a different legality definition. The first legality requirement of Definition 5.2 is exactly the same as in [MCF$^+$11], but the second requirement was not stated before. Indeed, we have only discovered during the work reported here that the second requirement is needed in order to give a clean semantics to precise derived-from edges. Finally, the OPM specification has an additional condition that we do not need here: there may be no cycles in the was-derived-from edges. We further discuss the implications of cycles in derived-from edges in OPM graphs in Section 5.4.2.

Note A.4.6. OPM introduces an observation interval that we do not formalize here. Our only assumption about time-points is that they can be partially ordered.

Note A.4.7. The OPM reference specification associates the create time-point with edges of the type `WasGeneratedBy`, since different accounts can assign different time-points to the same edge. In the presence of an edge of the type `WasControlledBy`, beginning and ending times of a process are associated with the edge.

Note A.4.8. The OPM reference specification defines a used-edge as meaning that the process required the availability of the artifact to be able to complete its execution: this is exactly Axiom 6. Likewise, the reference specification defines a was-generated-by edge as meaning that the process was required to initiate its execution for the artifact to have been generated: this is exactly Axiom 5. The OPM reference specification defines a was-triggered-by edge exactly as Axiom 7.

In addition, OPM time constraints indicate that the use time-point follows the start of the process, and the generate time-point precedes the end of the process. The OPM reference specification does not cover the time constraints related to multi-step edges of the types `Used*` and `WasGeneratedBy*`. It is a contribution of this work to have integrated all these constraints into a single set of axioms.

Note A.4.9. The OPM reference specification is not specific about the difference between the precise was-derived-from edge $A \xrightarrow{!} B$ and its imprecise version $A \rightarrow B$. First, a precise edge $A \xrightarrow{!} B$ can only exist in the presence of a generate–use–derive triangle, according to Definition 5.2. Second, both the create and use time-points (for A and B, respectively) occur within the scope of the process identified by the triangle. These constraints do not hold for $A \rightarrow B$.

Note A.4.10. Technically, the OPM reference specification defines refinement between two accounts. However, like graph operations, it is useful to define refinement over OPM graphs.

Note A.4.11. The OPM reference specification defines three completion operations: process and artifact introduction, as in this work, but also artifact elimination. Artifact elimination is precisely the inference of *informed-by* in Definition 5.13.

Note A.4.12. The OPM reference specification defines edges as *causal* relationships, but does not provide an explanation for this terminology, and the kind of causality underpinning them. In this work, we have chosen not to adopt this terminology to avoid unnecessary technical jargon. However, our temporal semantics provides a clarification with regard to this notion. Indeed, the kind of causality underlying OPM relationships is the causality typically defined in distributed systems. The constraints $u \preceq v$ in our temporal theory are similar in nature to Tel's causal order [Tel94, Definition 2.20]. In particular, according to Tel, a send event precedes the corresponding receive event in a distributed system, very much like our create time-point precedes a use time-point for the same artifact. Similar orders were referred to as "causality relation" by Mattern [Mat89] and "happened before" by Lamport [Lam78].

B

Samenvatting

Een workflow is een beschrijving van een complex en mogelijk langdurig proces, dat bestaat uit verschillende taken die moeten uitgevoerd worden in een gegeven volgorde [vdAvH04, SGBB01, BF05, FKSS08]. Taken bestaan uit andere taken, wat aanleiding geeft tot control-flow afhankelijkheden. De control flow hoeft niet lineair te zijn: sommige taken kunnen tegelijk, of alternatief, uitgevoerd worden. Naast de control flow is er ook een data flow. Het is belangrijk een onderscheid te maken tussen een workflow-specificatie enerzijds, en een workflow-uitvoering anderzijds. Een workflow-specificatie wordt ook een *workflow template*, *process definition*, of, naargelang het niveau van abstractie, zelfs *executable workflow* genoemd. In een onderneming kunnen vele workflow-specificaties gebruikt worden, en elke specificatie kan vele uitvoeringen hebben. Een workflow-management systeem ondersteunt het ontwerp van workflow-specificaties en beheert de verschillende uitvoeringen.

Workflow management is ontstaan in business process modeling [vdAvH04], maar recent is het belang van workflows in e-Science toegenomen, tesamen met de opkomst van Grid Computing [FK04]. Alhoewel het onmogelijk is een harde scheidingslijn te trekken tussen business workflows en scientific workflows, ligt de nadruk bij scientific workflows toch voornamelijk op data flow, en minder op control flow [SKDN05, GDE$^+$07, FKSS08]. Om deze reden spreken we ook over scientific workflows als *dataflows*.

Het gebruik van workflows in e-Science beslaat volgende aspecten:

- ontwerp en uitvoering van dataflows;
- opslag van dataflows en verwante gegevens in een repository;
- ondervragen van dataflow repository's;

- sharing van dataflow-gegevens met derden.

Dataflow repository's dienen vele doeleinden:

- Het efficiënt beheer van de vele experimentele en workflow gegevens die bestaan in een laboratorium of onderneming, helpt het volgen van de "scientific method" [Bro08].
- Controle van resultaten, hetzij intern, hetzij bij peer review, hetzij door derden, ter reproductie van de resultaten. Deze controle vereist in vele gevallen het opsporen van de herkomst ("provenance") van data waarden, of onderdelen daarvan, die voorkomen in het resultaat van de workflow.
- Beschikbaar stellen van de gegevens, inclusief de dataflows, ter beantwoording van decision-support vragen. Het aantal mogelijke vragen is eindeloos; we geven slechts twee voorbeelden:
  - "Gaf een vroegere uitvoering van de dataflow, die gebruik maakte van een oudere versie van GenBank, hetzelfde gen als resultaat?",
  - "Hebben we ooit een dataflow uitgevoerd waarin deze sequentie werd opgezocht met behulp van BLAST ?".

Een formeel model van dataflow repository's Het eerste doel van ons werk is het bijdragen van een formeel, conceptueel datamodel waarin een aantal van de belangrijkste aspecten van dataflow management worden samengevoegd. Verschillende workflow-management systemen en provenance-management systemen* zijn reeds ontwikkeld, dikwijls bovenop general-purpose databases.† De gebruikte databases beslaan het volledige gamma: (i) key-value [HSBMR08, MGM$^+$08], (ii) relationeel [CM95, AIL98, SKDN05, BD08, GCM$^+$11, FSC$^+$06, SPG08, KDG$^+$08, MGM$^+$08], (iii) XML [FSC$^+$06, SPG08, FMS08], (iv) RDF [KDG$^+$08, ZGST08], of (v) file-based [LAB$^+$06, MGM$^+$08].

Er is echter geen standaard conceptueel model onderliggend aan zulke systemen. Een dergelijk model zou tenminste volgende aspecten moeten beslaan:

*Een provenance-management systeem voor workflows registreert enkel informatie over de uitvoeringen van workflows. Zo'n systeem laat het gebruik van scripts, general-purpose programmeertalen en hogere-orde workflow talen toe.

†Sommige systemen gebruiken verschillende databases voor de specificaties enerzijds, en voor de uitvoeringen anderzijds.

- Een datamodel voor de complexe datastructuren die dienen als input voor een dataflow, of die worden geproduceerd als output, of die voorkomen als tussenresultaten.
- Een formele operationele semantiek voor de operatoren die het systeem ter beschikking stelt om elementaire transformaties uit te voeren op de complexe datastructuren.
- Een formele definitie van de informatie die vervat zit in de uitvoering van een dataflow. Deze informatie wordt dikwijls de *workflow provenance* [CT09, MCF$^+$11] of ook de *retrospective provenance* [FKSS08] van de dataflow-output genoemd.
- De rol die externe services spelen in een dataflow-uitvoering. Externe services worden typisch geleverd door derden, en het dataflow-management systeem voorziet een wrapperfunctie. Deze externe services vormen de elementaire taken van een scientific workflow.
- Het gebruik van een dataflow als taak in een andere dataflow.
- Een conceptueel model voor een repository bestaande uit meerdere dataflow-specificaties tesamen met hun verschillende uitvoeringen.
- Het ondervragen van zulke dataflow repository's, in het bijzonder wat betreft de nauwkeurige herkomst van onderdelen van dataflow-outputs. Dit type vragen staat bekend als *where-provenance*.

Een conceptueel datamodel voor dataflow repository's moet een preciese beschrijving geven van de gegevensstructuren vervat in de repository, met inbegrip van de dataflows zelf, en van de verbanden tussen deze gegevens. Wij vinden ons doel belangrijk omdat het een formeel kader biedt waarbinnen het volgende mogelijk wordt:

- De rigoureuse analyse van de mogelijkheden en de beperkingen van de verschillende systemen gebruikt in de praktijk, alsook het vergelijken van deze systemen.
- Het naar boven brengen van mogelijke semantische verschillen in gebruik van veelgebruikte begrippen zoals "workflow", "provenance", of "collection".

We stellen duidelijk dat ons doel er *niet* in bestaat een blauwdruk voor te stellen voor een nieuw dataflow-management systeem, met meer of betere features dan de "concurrentie". Ons werk is wel een gedetailleerde poging om zulk een systeem te modelleren, in de hoop hiermee een synthese bij te dragen van een aantal belangrijke database-aspecten van dataflow-management systemen. Het moge duidelijk zijn dat elk van deze aspecten in mindere of meerdere mate reeds ondersteund wordt door sommige systemen gebruikt in de praktijk. In het ideale geval zou er een standaard database-schema en een repository bestaan, voor de uitwisseling van dataflow-specificaties en uitvoeringen, naar analogie met het myExperiment.org initiatief dat specifiek is voor het Taverna systeem. De nood voor zo'n uitwisselingssysteem werd ook benadrukt door Lacroix [KCL06, KLL$^+$07]. Ons werk is een nieuwe stap in die richting, nadat de eerste stappen reeds in de jaren 1990 gemaakt werden, maar te weinig navolging kregen [CM95, AIL98]. De nood aan workflow repository's is ook benadrukt in andere wetenschapsgebieden, waarbij Blockeel en Vanschoren [Blo06, BV07] een pioniersrol spelen met de Experiment Databases for Machine Learning.

Dataflow-management systemen zijn voornamelijk onderzocht binnen de onderzoeksgemeenschap in computer systems, en minder in de onderzoeksgemeenschap van database theory. We hopen deze lacune gedeeltelijk te vullen door ons werk. We merken op dat er binnen database theory wel degelijk veel aandacht wordt besteed aan de statische analyse van data-intensive workflows, maar die onderzoeksrichting staat duidelijk loodrecht op de nadruk op modelleren en ondervragen die we leggen in dit werk.

Voor overzichten van workflow-management en provenance-management systemen verwijzen we naar Freire et al. [FKSS08], Yogesh et al. [SPG05], Bose and Frew [BF05], en Davidson and Freire [DF08]. Where-provenance is een vorm van "data provenance" onderzocht in database-onderzoek [BKT01, CT09, CCT09]. In ons werk tonen we aan hoe where-provenance kan gedefinieerd worden in de context van workflow provenance.

Semantiek voor het Open Provenance Model Het tweede doel van ons werk is een bijdrage te leveren aan de semantiek van het Open Provenance Model (OPM) [MCF$^+$11]. OPM is een systeem-onafhankelijk formaat voor de uitwisseling van provenance gegevens. OPM beantwoordt hierin aan dezelfde nood die ons inspireert in het eerste deel van ons werk; OPM legt zich uitsluitend toe op uitvoeringen van allerhande processen, maar beperkt zich daarbij niet noodzakelijk tot business of scientific workflows. OPM fungeert als gemeenschappelijke taal in een project van de W3C Provenance Incubator Group[‡] waarin men de concepten gebruikt in verschillende voorstellen van provenance modellen voor Semantic Web technologieën vertaalt in een ge-

[‡]`http://www.w3.org/2005/Incubator/prov/wiki/Main_Page`

meenschappelijk model, OPM dus. Zopas is een W3C Provenance Working Group[§] opgericht, gebaseerd op het werk van de W3C Provenance Incubator Group, en deze Working Group zal een standaardtaal voor uitwisseling van provenance-gegevens definiëren.

De huidige specificatie van OPM [MCF$^+$11] is in hoofdzaak syntactisch. Nochtans wordt een semantiek gesuggereerd, zowel in woorden als in de vorm van een aantal inferentieregels. Wij stellen een formele, temporele semantiek voor OPM voor, en geven een volledige set inferentieregels voor temporele inferentie in OPM. Dit werk gebeurde in samenwerking met Luc Moreau [KMV10].

Andere onderzoekers hebben ook reeds voorstellen gedaan voor een semantiek voor OPM.

Cheney [Che10] onderzoekt structurele causale modellen als semantiek voor provenance-grafen, en brengt een aantal OPM concepten in verband met de begrippen van actual cause en explanation voorgesteld door Halpern en Pearl [HP05]. Een gelijkenis tussen onze temporele semantiek en die van Cheney is dat beide voorstellen een wiskundige betekenis verbinden aan OPM grafen; er zijn echter ook belangrijke verschillen. (i) Onze semantiek voldoet aan letter en geest van de OPM reference specification [MCF$^+$11] en behandelt tijd, alle soorten edges, algebraïsche operatoren, en refinement. Cheney beschouwt derived-from edges als afleidbaar, wat niet de bedoeling is van OPM. (ii) Onze semantiek is zuiver temporeel, terwijl Cheney een OPM-graaf beschouwt als een functie van inputs naar outputs. Dit zijn twee complementaire visies. (iii) De semantiek van Cheney heeft een ander doel, namelijk, van een gegeven uitvoering het onderliggend programma (of een benadering daarvan) te recupereren.

Missier en Goble [MG11] behandelen de vraag of, gegeven een OPM graaf, er een plausibele workflow in Taverna kan gedefinieerd worden, die de gegeven graaf zou kunnen genereren. Daartoe is in het algemeen extra informatie nodig. Wanneer een OPM-graaf kan vertaald worden in een Taverna workflow, geeft dit indirect een semantiek van de OPM-graaf, via de bekende trace-semantiek voor Taverna [SH09a, SH09b, SHMG10]. Deze aanpak is interessant maar duidelijk niet volledig.

B.1 Korte inhoud

In Hoofdstuk 2 definiëren we ons dataflow-model voor complexe objecten [HKS$^+$07, HKS$^+$08]. Ons model is inderdaad sterk gebaseerd op het bekende

[§]`http://www.w3.org/2011/prov/wiki/Main_Page` complex-object datamodel. Als taal waarin de dataflow-specificaties worden geschreven, nemen we de Nested Relationele Calculus (NRC). Deze taal dient als een abstractie voor de elementaire operaties op complexe objecten binnen een dataflow. Deze operaties kunnen aangeboden worden door het workflow-systeem als taken, en zijn nuttig voor het converteren tussen dataformaten, of voor het formatteren van data voor output- of presentatiedoeleinden.

Verschillende workflow-management systemen ondersteunen reeds complexe objecten, zoals Taverna [TMG$^+$07, MBZ$^+$08], Taverna 2 [SHMG10], Kepler-CoMaD [MBL06, BML08], en Pegasus-Chimera [FVWZ02, CFV$^+$08]. Een typische operatie is deze die een gegeven functie toepast op alle elementen van een verzameling.

Uiteraard kunnen vele operatoren gebruikt worden om de basistaken binnen een dataflow met elkaar te verbinden. In ons werk concentreren we ons op operatoren voor de transformatie van complexe objecten, en beperken ons daarbij tot de welbekende operatoren van de NRC. Dit motiveert onze keuze voor NRC, uitgebreid met taken, als taal voor dataflow-specificatie.

De geschiktheid van NRC (in de variant CPL) voor gegevensmanipulatie en gegevensintegratie in wetenschappelijke toepassingen werd reeds uitvoerig gedemonstreerd door het Kleisli systeem [CCW03, DW04]. Wij hebben deze geschiktheid verder bevestigd via een aantal eigen case studies, ook gebaseerd op gepubliceerde protocollen uit de bioinformatica [GJZ06, ŁKO$^+$06, NDK$^+$06, GV04]).

In Hoofdstuk 3 definiëren we een conceptueel model voor dataflow repository's.

- We geven formele definities van uitvoeringen, *runs* genaamd, van NRC dataflows.
- We formaliseren de voorstelling van externe diensten in een dataflow repository.
- We formaliseren hoe abstracte taken in een dataflow-specificatie kunnen vervangen worden, hetzij door externe diensten, hetzij door het oproepen van andere dataflows.
- We stellen een eenvoudig conceptueel schema voor van een dataflow repository, en formuleren een aantal essentiële integrity constraints.
- We beschrijven een proof-of-concept voorstelling in SQL:2003.

In Hoofdstuk 4 bespreken we het ondervragen van dataflow repository's. Over het algemeen kunnen query's uiteraard geprogrammeerd worden in SQL/XML.

Daarbij echter identificeren we *subvalue provenance* als een fundamenteel query-onderdeel, en we geven een formele uitwerking van dit begrip. Deze query berekent de where-provenance van een gegeven deelwaarde van de output van een gegeven run. Ook formaliseren we de *subruns* van een run. Dit laat ons toe de where-provenance te bepalen van deelwaarden op elk niveau van de dataflow.

In verscheidene workflow-management systemen wordt een standaard query-taal gebruikt voor het ondervragen van de repository [MLA+08, FKSS08, SGM11]:

- REDUX [BD08], Swift [GCM+11] en Pegasus [KDG+08] gebruiken SQL,
- Taverna [ZGST08] en Pegasus gebruiken SPARQL, en
- ES3 [FMS08] en PASOA/PreServ [MGM+08] gebruiken XQuery.

Jammer genoeg gebruiken al deze systemen een verschillend logisch database-schema; enkel voor uitwisseling van gegevens kan het gemeenschappelijk OPM-formaat gebruikt worden.

We vermelden nog het interessante werk van Beeri et al. [BEKM08] betreffende het ondervragen van de *mogelijke* uitvoeringen van een gegeven workflow-specificatie. Deze aanpak is eerder gebaseerd op verification en legt minder de nadruk op repository's. Er werd wel gekeken naar monitoring [BEMP07].

Tenslotte, in Hoofdstuk 5, geven we een formele, temporele semantiek voor het Open Provenance Model. Onze hoofdbijdrage bestaat in een set van inferentieregels die werken op het logisch niveau van OPM-grafen, en die sound en complete zijn voor logische gevolgtrekking uit de temporele axioma's verbonden aan een OPM-graaf.