OPEN ACCESS

KING'S College LONDON

King's Research Portal

DOI:
10.1504/IJBPIM.2010.033174

Document Version
Peer reviewed version

Link to publication record in King's Research Portal

Citation for published version (APA):
Curcin, V., Ghanem, M., & Guo, Y. (2010). Polymorphic type framework for scientific workflows with relational data model. *International Journal of Business Process Integration and Management*, *5*(1). https://doi.org/10.1504/IJBPIM.2010.033174

Citing this paper
Please note that where the full-text provided on King's Research Portal is the Author Accepted Manuscript or Post-Print version this may differ from the final Published version. If citing, it is advised that you check and use the publisher's definitive version for pagination, volume/issue, and date of publication details. And where the final published version is provided on the Research Portal, if citing you are again advised to check the publisher's website for any subsequent corrections.

General rights
Copyright and moral rights for the publications made accessible in the Research Portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognize and abide by the legal requirements associated with these rights.

•Users may download and print one copy of any publication from the Research Portal for the purpose of private study or research.
•You may not further distribute the material or use it for any profit-making activity or commercial gain
•You may freely distribute the URL identifying the publication in the Research Portal

Take down policy
If you believe that this document breaches copyright please contact librarypure@kcl.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 14. Oct. 2024

Polymorphic type framework for scientific workflows with relational data model

Vasa Curcin
Department of Computing,
Imperial College London, United Kingdom
E-mail: vc100@doc.ic.ac.uk
*Corresponding author

Moustafa Ghanem
Department of Computing,
Imperial College London, United Kingdom
E-mail: mmg@doc.ic.ac.uk

Yike Guo
Department of Computing,
Imperial College London, United Kingdom
E-mail: yg@doc.ic.ac.uk

Abstract: Scientific workflow systems provide languages for representing complex scientific processes as decompositions into lower level tasks, down to the level of atomic, executable units. To support data analysis activities, a wide variety of such languages represent data transformation and processing operations as task nodes within a workflow. Adding data type information to the task inputs and outputs allows workflow authors to perform type checking at design time, search for compatible nodes in public component repositories and define specifications of abstract workflows. Introducing support for strict data typing simplifies the implementation of a workflow system in addressing these issues, but at the expense of losing flexibility. We address this challenge by developing a data typing framework for scientific workflow systems that supports polymorphic data types that specify the minimal type constraints on node inputs and outputs. We focus on applications that use a relational data model and provide a polymorphic type formula composition algorithm for workflow nodes and fragments. The techniques introduced are validated by applying the inference engine prototype to an adverse drug reaction study performed with the relational algebra subset of the Discovery Net workflow system.

Keywords: scientific workflow; dataflow; type inference.

Reference to this paper should be made as follows:

1 Introduction: Scientific workflows

Over the past decade the workflow paradigm has been advocated as a formalism for the definition of scientific processes and also as a basis for developing user-oriented e-Science (Hey and Trefethen, 2002) application development and execution systems. Workflows naturally provide a visual process representation in terms of dependencies between tasks, capture structure of data integration activities(Mahoui et al., 2005; Ludäscher and Raschid, 2005) and effectively serve as an orchestration layer in service oriented architectures where each task represents a remotely accessible data or computational service (Xu et al., 2004).

These features led to the coining of the term *scientific workflow* (Taylor et al., 2006; Ludäscher and Goble, 2005; Gil et al., 2007; Chen and van der Aalst, 2007), as a type of workflow focused on supporting the scientific research process through several aspects. First, allowing ad-hoc construction and execution of workflows that access remote data sources and scientific instruments, envisaging that the workflow will progress through a series of explorative, intermediate designs, each of which is investigated and tested by the user in an interactive manner. Second,

Copyright © 200x Inderscience Enterprises Ltd.

providing explicit support for collaboration between distributed users, by using the workflows as means of communicating research ideas. Finally, supporting a visually intuitive user interface that does not require a high level of technical expertise from the user.

The popularity of the paradigms has led to the design and implementation of a wide variety of specialized scientific workflow systems. Examples include Discovery Net (Ghanem et al., 2008), Taverna (Hull et al., 2006), Kepler (Altintas et al., 2004) and Triana (Taylor et al., 2005). It has also led to the surge in the use of other workflow-like systems in scientific data analysis. Examples include InforSense[1], PipelinePilot[2], KNIME (Berthold et al., 2007), Orange (Demšar et al., 2004), Pentaho[3] and, more generally, workflow front ends for existing data analysis tools such as SPSS Clementine[4] and SAS Enterprise Miner[5].

With the increase in usage and popularity of scientific workflows, so have emerged the various supporting technologies, intended to make the workflow creation easier and faster. First and foremost were graphical workflow composition tools that enabled non-technical users to create meaningful applications and providing them with interactive visual data analysis and expection tools. Another advance were node and workflow repositories, such as myExperiment (Roure et al., 2007), which store user-submitted workflow fragments and full workflows, allowing the user to reuse existing work and tailor it to their purpose. Finally, incremental collaborative design is also realized through supporting the design of abstract workflows.

1.1 Motivation

Since scientific data and results are the main curency of scientific explorations, large fragments of scientific workflows are typically data flows with data flowing through data transformation and analysis tasks. A key challenge in many generic workflow systems thus becomes dealing with data type mismatches within a workflow. If no data type information is present, a user can construct a workflow that looks perfect to her task, only to discover a runtime error after several hours of execution when one of the tasks in the workflow receives data of the wrong type.

Adding data type information to the inputs and outputs of the individual workflow nodes can alleviate this problem by allowing design time type checking and preventing errors occurring at runtime, thereby saving effort and computational time. Essentially, typing allows us to answer two questions:

- Can a given workflow be applied to a given input?
- Given two workflow components or fragments and two ports belonging to them, can the two components be composed by creating a link between those two ports?

In strictly typed workflow systems the output of one workflow node cannot connect to the input of another except if the data types match *exactly*. However, such strict typing reduces the flexibility in workflow definitions and reuse when connecting new data sources and implementations. Typically, a workflow component has a set of required attributes that its input need satisfy, and as long as a *partial* match is present, it is happy to execute.

Polymorphic typing (Milner, 1978) provides the solution, by defining the type transformation associated with each node, and separating node inputs into the *fixed* part, which node requires to be present, and the *variable* part, that may or may not be present in the output. Therefore, the type formula associated with the workflow declares the fixed and variable parts of the output for any given input, allowing polymorphic typing to answer further two questions:

- Given a workflow type formula and new input data type, what will be the exact output type produced by the workflow?
- Given a workflow composed of nodes with given type formulas, what will be the type transformation defined by the composition?

Benefits of polymorphic approach in workflow systems are manifold. Design time type checking prevents errors occurring at runtime and saves effort and computational time. Adding type information to nodes in a repository allows automatic matching of a data source to available nodes. Similarly, providing the type specification of a context hole in an abstract workflow facilitates the grounding of those workflows with concrete components.

In this paper, we focus on the relational data model in scientific workflows, where basic data units are tables consisting of sets of records of attribute-value pairs. While often overlooked in favour of XML-based models that are targeted towards SOAP web service composition, this category covers a large number of analysis-oriented workflow systems and data analytics tools. By mapping directly onto the relational database schemas, the relational workflow model is particularly suited to representing data transformations, Extract-Transform-Load tasks and data mining compositions.

1.2 Layout

Section 2 presents a generic relational workflow system with the basic operations. Section 3 introduces the polymorphic type formula framework and concepts of type transformations and type formula composition. The techniques introduced are validated in section 4 by applying the prototype inference engine to an adverse drug reaction workflow in Discovery Net data flow system to demonstrate the workings of the framework. Paper finishes with conclusions and directions for future work.

[1] InforSense, www.inforsense.com
[2] Pipeline Pilot, http://accelrys.com/products/scitegic/
[3] Pentaho, www.pentaho.com
[4] SPSS Clementine, www.spss.com/clementine
[5] SAS Enterprise Miner, www.sas.com/em

2 Relational workflow system

As discussed in the previous section large fragments of scientific workflows are simply data flows, where the data flows between nodes representing data transformation operations with the input of one operation serving as input for another. The data properties of such workflow can be represented by:

1. Data domain $\mathcal{D}$ which may contain a subset of constant symbols, denoted A.
2. A set of function labels $\mathcal{F}$ corresponding to functions of form $\mathcal{D}^k \rightarrow \mathcal{D}$, where k is the cardinality of the function.
3. Semantic function $\mathcal{S}[\![]\!]$ which performs the valuation of rooted subgraphs.
4. A set of rewrite rules that define graph equivalences in $\mathcal{D}$.
5. A type formula for each of the functions in $\mathcal{F}$.

In this paper, we will mainly concern ourselves with data domain $\mathcal{D}$, functions in $\mathcal{F}$ and their corresponding type formulas. The semantic function $\mathcal{S}[\![]\!]$ produces the output value of a node given a full set of inputs, and rewrite rules are used to define workflow equivalencies for purpose of optimization, as detailed in Curcin (2009). Note that a pure data flow execution model is assumed (Curcin et al., 2007), in which execution ordering is defined by the data dependencies of the node whose result is requested.

The data structures in $\mathcal{D}$ correspond to elements in Codd's relational algebra (Codd, 1970), and represent relations over sets of tuples – ordered domain/value pairs. Two example relations, taken from Date (1995) are presented in Figure 1 with relation S containing suppliers, and SP containing mappings between suppliers and the products they supply. The data in relational model is represented

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SP

S#	P#
S1	P1
S1	P2
S1	P3
S1	P4
S1	P5
S1	P6
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5

Figure 1: Two database relations

using n-ary relations, which are subsets of Cartesian product of n types $\tau_1, \tau_2, ...\tau_n$. Each relation has a heading and a body, where heading is a set of ordered pairs of attribute names and types associated with those names, and body is a set of n-tuples, each of which is an unordered set of attribute name–value pairs[6].

[6]Some adaptations of the relational model impose ordering on the set, but it is irrelevant for our purposes.

We also define eight function labels, and $\mathcal{F} = \{Delete, Select, Join, Filter, Group, Difference, Union, Derive\}$, each with a corresponding function mapping relations to other relations. This set of components allows formulation of numerous data analysis tasks and will be the basis of a case study in section 4.

Delete has a parameter specifying which columns to delete. Using the example relation S:

$$Delete[\text{STATUS}]S = \{[\{(S1, Smith, London), (S2, Jones, Paris), ... (S5, Adams, Athens)\}\}]$$

Select is the exact opposite, in that it retains only the given attributes:

$$Select[\text{STATUS}]S = \{[\{20, 10, 30, 20, 30\}\}]$$

The parameter associated with filtering is a propositional logical expression defined over the attributes, and its result is a relation with the same heading as the input, but only including the tuples for which the expression evaluates to true.

$$Filter[\text{STATUS} = 20]S = \{(S1, Smith, 20, London)\}$$

Difference calculates the difference of two inputs. Unlike the classical relational variety, which operates on two identical input tables, the version presented here is based on a single identifier column. The rows preserved in the first table are the ones that do not contain identifier values present in the second table.

Join (equi-join) has one parameter which specifies the column to perform the join on. Group specifies the grouping column, and the aggregation functions to perform on the grouped attributes. Derive takes a column name, an expression, and adds the column containing the expression evaluation to the relation.

The functions in $\mathcal{F}$ are defined using the standard operators of the relational algebra with added aggregation (γ) and extended projection ($\pi_{f(X)}$).

$$\begin{aligned}
&Union \quad X \quad Y \longrightarrow_\delta X \cup Y \\
&Difference[M] \quad X \quad Y \longrightarrow_\delta (\pi_M X \backslash \pi_M Y) \times X \\
&Join[M] \quad X \quad Y \longrightarrow_\delta \tau_{X.M=Y.M}(X \times Y) \\
&Delete[M] \quad X \longrightarrow_\delta \hat{\pi}_M X \\
&Select[M] \quad X \longrightarrow_\delta \pi_M X \\
&Filter[M] \quad X \longrightarrow_\delta \tau_M X \\
&Group[GA, AL] \quad X \longrightarrow_\delta \gamma_{GA,AL}(X) \\
&Derive[A, M] \quad X \longrightarrow_\delta \pi_{X,M}
\end{aligned}$$

An example relational workflow is shown in Figure 2, representing the calculation of parts that are available in Paris with status 10. Tables S and SP are used as inputs and the nodes are as defined above.

Note that there are multiple flavours of relational operators and the nodes selected are not the only ones possible.

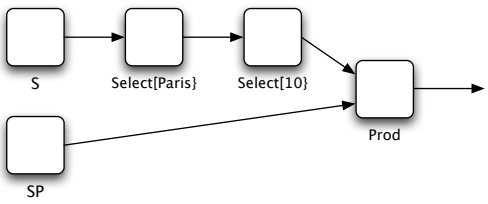

Figure 2: Relational workflow example

For example, Union and Join could be designed to take multiple inputs, explicit Product node could replace Join, etc. however, this has no impact on the type compositions presented here which are generic. These particular nodes have been selected as they appear in Discovery Net relational segment and have been used for the prototype type inference engine and the case study in section 4.

3 Type polymorphism in scientific workflows

If the data flow defines explicit types for its data model, each node has with it an associated type transformation that is defined by its abstract type formula. Furthermore, any composition of nodes, also has a type formula defined from the formulas of the internal nodes. In this section we show how these formulas are constructed.

An advantage that type systems offer is to allow polymorphism in workflow definitions – define the constant (required) and variable parts in both input and output. In a relational example, the constant is a column in the data set that is needed for the operation, in an XML-like language, it could be a tree branch. The variable part is everything else in the data type.

As a simple example, consider a data flow operating on relational tables. Let us also define a component $\sigma_{A>3}$ corresponding to a relational calculus operation which only preserves the table rows with attribute A having value greater than 3. The minimum requirement for the input data table to the component is to have the column A present, and it constitutes the minimal part of the input type. Therefore, a table with header (A, B, C) would satisfy the composition, but table (B, C, D) would not. The polymorphic part of the type in the former is (B, C), which is preserved by the type formula.

The type formalism of programming language structures is an established field. The first algorithm for type inference in functional programming languages can be found in Milner (1978) which was later investigated for other types of languages in MacQueen et al. (1984). Polymorphic type inference for relational calculus was studied in van den Bussche and Waller (1999), which also introduced the syntax for type formulas used in this section for data flow constructs.

3.1 Type systems

We define a finite set $\mathcal{B}$ of base types, and a set $\mathcal{T}$ of all types, with an associated set of complex type constructors (sets, lists, maps, tuples, records...).

Definition 3.1. Given a finite set $\mathcal{B}$ of base types, set C_S of shallow type constructors, $c_S : 2^{\mathcal{B}} \longrightarrow \mathcal{T}$, and set C_N of nested type constructors $c_N : 2^{\mathcal{T}} \longrightarrow \mathcal{T}$, type system $\mathcal{T}$ is the minimum set satisfying the following:

- $\perp \in \mathcal{T}$. Type for empty set is always present.
- $\mathcal{B} \subseteq \mathcal{T}$. All base types are types.
- If $\tau \in \mathcal{B}$ is a type, then $c_S(\tau) \in \mathcal{T}$ is also a type.
- If $\tau \in \mathcal{T}$ is a type, then $c_N(\tau) \in \mathcal{T}$ is also a type.

Given some set of data variables, the type assignment $\mathcal{A}$ maps each variable to its type. The notion that a variable a has type τ under $\mathcal{A}$ will be written as $\mathcal{A} \vdash a : \tau$.

We will restrict ourselves to types that have the notion of subtype present, in the sense that if $\tau_1 \subseteq \tau_2$ then $\mathcal{A} \vdash a : \tau_1 \Longrightarrow \mathcal{A} \vdash a : \tau_2$ for all a. The subtyping relation is dependent on the data model – in relational calculus it denotes that the set of columns defined by τ_1 is a subset of columns defined by τ_2, in an XML-like nested structure, it denotes that τ_1 describes a subtree of the structure defined by τ_2.

In the remainder of the section, the relational model will be used to demonstrate how a type system can be applied to workflows. The set of shallow constructors is then $C_S = \{Tuple\}$ with $Tuple$ instances written as $\langle l_1 : \tau_1, l_2 : \tau_2 ... \rangle$, where $l \in \mathcal{L}$ is the set of labels, and $C_N = \{Set\}$. Since all the variables considered will be sets of tuples, all type variables will be taken to denote a set. The subtype relation between two type variables will denote that the tuple attributes in one tuple are the subset of attributes in the other.

Definition 3.2. (Type context) Type context Γ is a quintuplet $\Gamma = (vars, tvars, att, dec, cons)$, where $vars$ are the data variables representing the labelled ports of the workflow or a component, $tvars$ denoted by t_i are the variables representing the types. att are the type constants (non–polymorphic parts) of input variables. dec are declarations mapping the variables to type variables representing their polymorphic parts and $cons$ is the mapping from typed constants onto propositional logic formulas over $vars$.

An instantiation of context Γ, denoted as $\mathcal{I}(\Gamma) : tvars \cup att \longrightarrow \mathcal{T}$ is a mapping which assigns to each type variable and type constant a set of data variables for which they hold. Two type variables in the context always represent disjoint types, $\mathcal{I}(t_1) \cap \mathcal{I}(t_2) = \emptyset$ and no type constant A in the context is present in an instantiation of a type variable in that context. Finally, the constraint on each property, $cons(A)$ is satisfied in the instantiation $\mathcal{I}$ if $\mathcal{I}(A) \vdash cons(A)$.

The type assignment $\mathcal{A}$ on each input data variable is defined in terms of the instantiation and the context as:

$$\mathcal{A}(x) := \bigcup_{t \in dec(x)} \mathcal{I}(t) \cup \{A \in att \mid x \in \mathcal{I}(A)\}$$

Definition 3.3. (Context compatibility) Two type contexts, Γ_1 and Γ_2 are said to be compatible, denoted $\Gamma_1 \equiv \Gamma_2$ iff there exists an instantiation function:

$$\mathcal{I}(\Gamma_1, \Gamma_2) : tvars_1 \cup tvars_2 \cup att_1 \cup att_2 \longrightarrow \mathcal{T}$$

such that:

1. For every $A \in att_1 \cap att_2$, it holds that $\mathcal{I}(A) \vdash cons_1(A) \wedge cons_2(A)$.
2. Any two distinct type variables still represent disjoint types, $\mathcal{I}(t_1) \cap \mathcal{I}(t_2) = \emptyset, t_1, t_2 \in tvars_1 \cup tvars_2$.
3. No type constant is present in an instantiation of a type variable.

The type formulas will be presented as the mapping between the input type context Γ_i and the output one Γ_o:

$$\begin{array}{ll} (vars) & (outvars) \\ (tvars) & (tvars) \\ (att) \quad \longmapsto & (att) \\ dec : vars \longrightarrow 2^{tvars} & outdec : outvars \longrightarrow 2^{tvars} \\ cons : att \longrightarrow \mathcal{L}_{vars} & outcons : att \longrightarrow \mathcal{L}_{outvars} \end{array}$$

dec and *outdec* map each input and output variable, respectively, to the set of type variables describing its variable part. $\mathcal{L}$ represents a propositional logic formula over the variables. The terms in brackets denote sets used in the declarations and will not be explicitly listed every time.

3.1.1 Example: Filter node

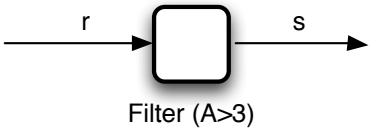

Figure 3: Filter typing example

Let us illustrate this notation by writing the type formula of the node $\sigma_{A>3}$, shown in Figure 3:

$$\begin{array}{ll} r : t & s : t \\ A : r \quad \longmapsto & A : s \end{array}$$

The variable part of the input type context r is assigned a type variable t, which is the polymorphic base for the operation. The type constant A represents the invariate part and is required to be present in any assignment on r.

Similarly on the output side, the context has a variable part and an invariate part. The variable part declares output variable s to have as its polymorphic part the type variable t and as its constant part the property A which, in all assignments, will be in s.

Definition 3.4. (Type formula) Type formula Φ associated with a node is the octuplet $\Phi = (tvars, att, vars, dec, cons, outvars, outdec, outcons)$, where *vars* and *outvars* are the input and output data variables of the node, respectively corresponding to input and output ports of n, *outdec* are mappings between those variables and *tvars*, and *outcons* maps the type constants in *att* to propositional formulas over *outvars* that define for which variables is the type constant satisfied.

Given the type context and the instantiation, the output variables of the formula have their type assignment defined as:

$$\mathcal{A}(x) := \bigcup_{x \in outvars} \mathcal{I}(x) \cup \{A \in att \mid \mathcal{I}(A) \vdash outcons(A)\}$$

3.1.2 Example: Delete node

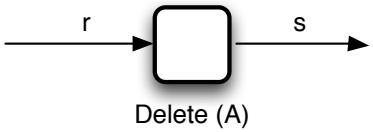

Figure 4: Typing of the Delete node

Consider the example of a *Delete* component shown in Figure 4[7]. Its function is to remove a column from the input data. Its type formula is:

$$\begin{array}{ll} r : t & s : t \\ A : r \quad \longmapsto & A : \neg s \end{array}$$

As shown, the variable part stays the same, but A has been removed from the output constraints.

3.1.3 Example: Join node

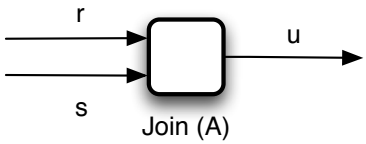

Figure 5: Typing of the Join node

Let us now consider a join node (Figure 5), which takes two inputs and produces the output with all the types of the input, unifying them on the attribute A.

$$\begin{array}{ll} r : t_1 & u : t_1 t_2 \\ s : t_2 \quad \longmapsto & A : u \\ A : r \wedge s & \end{array}$$

Notice how in order for the constraint for A to be satisfied, both r and s need to satisfy A. Therefore, the output is the concatenation of types from both nodes, with A guaranteed to be present.

[7] *Delete* is sometimes called inverse select, and denoted with $\hat{\pi}(A)$.

3.2 Composing type formulas

So far, examples of type formulas for individual components have been given. We will now look into how two nodes connected on a port define a joint type formula. This procedure is generic and will allow us to define the type formula for the entire workflow.

Connection between the output of one node and the input of another has two consequences for the type formulas.First, output variable of the source node equates to the input variable of the destination node. Second, a type mapping is established between the variables used to describe the polymorphic parts in the output and input, $outdec_1(x)$ and $dec_2(x)$.

Definition 3.5. (Composability of type formulas) As a consequence of this, two node type formulas Φ_1 and Φ_2 are composable on variable x, denoted $\Phi_1 \overset{x}{\Rrightarrow} \Phi_2$, if:

1. There is such instantiation $\mathcal{I}$ for all $x \in outvars_1 \cap vars_2$ that:
$$\bigcup_{t \in outdec_1(x)} \mathcal{I}(t) = \bigcup_{t \in dec_2(x)} \mathcal{I}(t)$$
In other words, each instantiation of the joint formula needs to ensure that the types represented by the type variables in both type formulas are identical.
2. Under that instantiation, for every $A \in att_2$, it holds that $\mathcal{I}(A) \vdash outcons_1(A) \wedge cons_2(A)$.

In the relational model, each node input variable is denoted with a single type variable, since there are no multiple outputs, and no need to partition the input types. Therefore, instead of adding constraints to the instantiation, it will be convenient to use the following function that maps the types of the second node to the types of the first as defined by the connection, or leaves them unchanged if they are not affected:

$$tmap(t) = \begin{cases} outdec_1(x) & \textbf{if } t \in dec_2(x)\textbf{,} \\ t & \textbf{if no such } x\textbf{.} \end{cases}$$

Furthermore, for each type formula Φ, we define a propositional formula $vmap_i(a)$ for each $a \in outvars_i$, such that:

$$vmap_i(a) = \bigvee_{x \in vars_i} x.dec_i(x) \in outdec_i(a)$$

Informally, the function translates each output data variable to a disjunction of input variables. We will use it to translate the type constant constraints to use the input variables, denoted $[vmap_1(x)/x]cons(A)$.

Finally, we define a similar function, $fmap(a, M)$ for translating a constraint on a variable from dec_2 into a conjunction of variables in $outvar_2$ that use its type variable.

$$fmap(a, M) = \begin{cases} \bigwedge\limits_{x \in outvars_2} & x.dec_2(a) \subseteq outdec_2(x) \\ & \quad \textbf{if } outcons_1(M) \vdash a\textbf{,} \\ \bigwedge\limits_{x \in outvars_2} & x.dec_2(a) \subseteq outdec_2(x) \wedge \\ & outcons_1(M) \vdash vmap_2(x) \\ & \quad \textbf{if } outcons_1(M) \vdash \neg a \end{cases}$$

Definition 3.6. (Type formula of a node composition) Two type formulas:

$$\begin{aligned} \Phi_1 = (&tvars_1, att_1, \\ &vars_1, dec_1, cons_1, \\ &outvars_1, outdec_1, outcons_1) \\ \Phi_2 = (&tvars_2, att_2, \\ &vars_2, dec_2, cons_2, \\ &outvars_2, outdec_2, outcons_2) \end{aligned}$$

that share link variables $\{v_i\} = outvars_1 \cap vars_2$ define a joint type formula Φ', as shown in Figure 6.

The key feature of the composition formula is how input and output constraints are combined. The input constraint of the new formula, $cons'$, will preserve the input constraints of the first node, but will only keep the input constraints of the second node (renamed using $vmap$) on the attributes that are not affected during the composition via variable v_i. If the first node, for example, deletes an attribute in its output, $outcons_1(A) \vdash \neg v_i$, the composition will not contain it anymore.

The second part of constraint calculation takes into account nodes with multiple input ports, in which connection is realized only on one port, but the typing constraints used must be propagated to the other input variables ($vars_2 \setminus \{v_i\}$). Two separate cases are distinguished for the variables that now have to possess a certain attribute because of the composition, and those that can't. Typical example is the union of two inputs that must have identical types: if the connection link specifies that one input must have attribute A, then the other input has to have it as well.

Output constraint formulas function in a similar way. The first part defines the constraints on attributes in the new composition, eliminating those that are removed in the connection, while the second part propagates the new type constraints for nodes with multiple outputs, and ensuring that any input constraint introduced in $cons_2$ will map onto another output from $outvars_1$ that remains free.

As an example, consider the workflow in Figure 7. An underlying relational data model is assumed. The variables are labelled r, s, u, the first node performs a filtering operation, while the second removes two columns from the input. Let us first see how *Filter* and *Delete* nodes are composed.

$$\text{Filter} ::= \begin{array}{c} r : t_1 \\ A : r \end{array} \longmapsto \begin{array}{c} s : t_1 \\ A : s \end{array}$$

$$\text{Delete} ::= \begin{array}{c} s : t_2 \\ A : s \\ B : s \end{array} \longmapsto \begin{array}{c} s : t_2 \\ A : \neg s \\ B : \neg s \end{array}$$

In our example, we start by establishing the $tmap$ mapping, which only has one value in both domain and the target: $tmap(t_2) = t_1$. Therefore, the polymorphic segments of types t_1 and t_2 are set to be identical.

$$
\begin{aligned}
\Phi' &= (tvars', att', vars', dec', cons', outvars', outdec', outcons') \\
vars' &= vars_1 \cup vars_2 \setminus \{v_i\} \\
tvars' &= [tmap(t_i)/t_i]tvars_1 \cup tvars_2 \\
att' &= att_1 \cup att_2 \\
dec'(x) &= \begin{cases} dec_1(x) & \textbf{if } x \in vars_1, \\ [tmap(t_i)/t_i]dec_2(x) & \textbf{if } x \in vars_2. \end{cases} \\
cons'_1(A) &= \begin{cases} cons_1(A) \wedge [\top/v_i]cons_2(A) & \textbf{if } outcons_1(A) \vdash v_i, \\ cons_1(A) \wedge [\bot/v_i]cons_2(A) & \textbf{if } outcons_1(A) \vdash \neg v_i, \\ cons_1(A) \wedge [vmap_1(v_i)/v_i]cons_2(A) & \textbf{otherwise.} \end{cases} \\
cons'_2(A) &= \bigwedge_{k \in vars_2 \setminus \{v_i\}} k. \begin{array}{l} dec_2(k) = outdec_1(v_i) \wedge \\ cons_2(A) \nvdash k \wedge cons_2(A) \nvdash \neg k \wedge \\ outcons_1(A) \vdash v_i \end{array} \\
&\quad \bigwedge_{l \in vars_2 \setminus \{v_i\}} \neg l. \begin{array}{l} dec_2(l) \subseteq outdec_1(v_i) \wedge \\ cons_2(A) \nvdash l \wedge cons_2(A) \nvdash \neg l \wedge \\ outcons_1(A) \vdash \neg v_i \end{array} \\
cons'(A) &= cons'_1(A) \wedge cons'_2(A) \\
outvars' &= outvars_1 \cup outvars_2 \setminus \{v_i\} \\
outdec'(x) &= \begin{cases} outdec_1(x) & \textbf{if } x \in vars_1, \\ [tmap(t_i)/t_i]outdec_2 & \textbf{if } x \in vars_2. \end{cases} \\
outcons'_1(A) &= \begin{cases} [\top/v_i]outcons_1(A) \wedge outcons_2(A) & \textbf{if } cons_2(A) \vdash v_i, \\ [\bot/v_i]outcons_1(A) \wedge outcons_2(A) & \textbf{if } cons_2(A) \vdash \neg v_i, \\ [fmap(v_i, A)/v_i]outcons_1(A) \wedge outcons_2(A) & \textbf{otherwise.} \end{cases} \\
outcons'_2(A) &= \bigwedge_{k \in outvars_1 \setminus \{v_i\}} k. \begin{array}{l} outdec_1(v_i) \subseteq outdec_1(k) \wedge \\ outcons_1(A) \nvdash k \wedge outcons_1(A) \nvdash \neg k \wedge \\ cons_2(A) \vdash v_i \end{array} \\
&\quad \bigwedge_{l \in outvars_1 \setminus \{v_i\}} \neg l. \begin{array}{l} dec_2(l) = outdec_1(v_i) \wedge \\ outcons_1(A) \nvdash l \wedge outcons_1(A) \nvdash \neg l \wedge \\ cons_2(A) \vdash \neg v_i \end{array} \\
outcons'(A) &= outcons'_1(A) \wedge outcons'_2(A)
\end{aligned}
$$

Figure 6: Type formula for composition of two nodes

$$
\begin{array}{ll}
vars' = \{r\} & \\
tvars' = \{t_1\} & outvars' = \{u\} \\
att' = \{A, B\} & outdec'(u) = t_1 \\
dec'(x) = t_1 & outcons'(A) = \neg u \\
cons'(A) = r & outcons'(B) = \neg u \\
cons'(B) = r &
\end{array}
$$

This is used as the starting point to define the other sets needed by the representation, and the next step to be undertaken is to construct the joint context from the new sets, producing the merged formula. The formula is shown here in the set form and as a schema.

$$
\begin{array}{lcl}
r : t_1 & & u : t_1 \\
A : r & \longmapsto & A : \neg u \\
B : r & & B : \neg u
\end{array}
$$

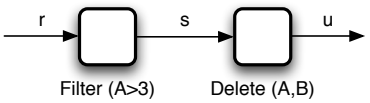

Figure 7: Sequential composition of types

3.2.1 Merge composition

Merge composition is used for nodes which combine data from multiple inputs, such as *Join* and *Union* operations in the relational calculus. The principle is the same as with sequential composition: checking the compatibility, and constructing the joint type formula. However, in the first step, the compatibility of the existing constraints from the input nodes also needs to be taken into account, since there may be shared variables involved.

We will consider the example in Figure 8. The three

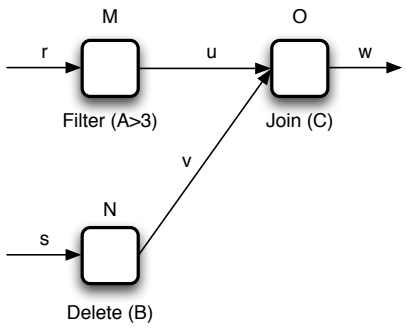

Figure 8: Merge composition of types

nodes have the following type formulae:

$$\text{M} := \begin{array}{l} r : t_1 \\ A : r \end{array} \longmapsto \begin{array}{l} u : t_1 \\ A : u \end{array}$$

$$\text{N} := \begin{array}{l} s : t_2 \\ B : s \end{array} \longmapsto \begin{array}{l} v : t_2 \\ B : \neg v \end{array}$$

$$\text{O} := \begin{array}{l} u : t_3 \\ v : t_4 \\ C : u \wedge v \end{array} \longmapsto \begin{array}{l} w : t_3 t_4 \\ C : w \end{array}$$

As a first step, let us join the M and O nodes. By composition, we have $tmap(t_3) = t_1$.

$$\begin{array}{l} vars' = \{r, v\} \\ tvars' = \{t_1, t_4\} \\ att' = \{A, C\} \\ dec'(r) = t_1 \\ dec'(v) = t_4 \end{array} \qquad \begin{array}{l} cons'(A) = r \\ cons'(C) = r \wedge v \\ outvars' = \{w\} \\ outdec'(w) = t_1 t_4 \\ outcons'(A) = w \\ outcons'(C) = w \end{array}$$

The resulting type formula has two input ports, one on M and one remaining on O:

$$\text{MO} := \begin{array}{l} r : t_1 \\ v : t_4 \\ A : r \\ C : r \wedge v \end{array} \longmapsto \begin{array}{l} w : t_1 t_4 \\ A : w \\ B : \neg w \\ C : w \end{array}$$

In the next step, the process is repeated to unify MO and N through link v. By connection, $tmap(t_4) = t_2$ and the type formula is defined as:

$$\begin{array}{l} vars' = \{r, s\} \\ tvars' = \{t_1, t_2\} \\ att' = \{A, B, C\} \\ dec'(r) = t_1 \\ dec'(s) = t_2 \end{array} \qquad \begin{array}{l} cons'(A) = r \\ cons'(B) = s \\ cons'(C) = r \wedge s \\ outvars' = \{w\} \\ outdec'(w) = t_1 t_2 \\ outcons'(A) = w \\ outcons'(B) = \neg w \\ outcons'(C) = w \end{array}$$

The final formula is:

$$\text{MNO} := \begin{array}{l} r : t_1 \\ s : t_2 \\ A : r \\ B : s \\ C : r \wedge s \end{array} \longmapsto \begin{array}{l} w : t_1 t_2 \\ A : w \\ B : \neg w \\ C : w \end{array}$$

3.2.2 Composition split

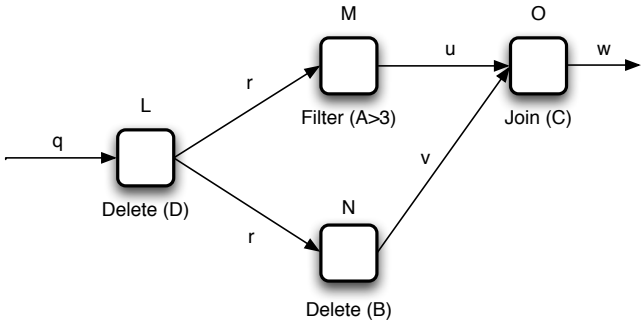

Figure 9: Split composition of types

The last case that needs to be considered is when one node is connected to two other nodes, as shown in Figure 9. We proceed as in the previous cases, joining first L and M.

$$\text{L} := \begin{array}{l} q : t_0 \\ D : q \end{array} \longmapsto \begin{array}{l} r1 : t_0 \\ r2 : t_0 \\ D : \neg(r1 \wedge r2) \end{array}$$

$$\text{M} := \begin{array}{l} r1 : t_1 \\ A : r1 \end{array} \longmapsto \begin{array}{l} u : t_1 \\ A : u \end{array}$$

$$\text{LM} := \begin{array}{l} q : t_0 \\ A : q \\ D : q \end{array} \longmapsto \begin{array}{l} r2 : t_0 \\ u : t_0 \\ A : u \wedge r2 \\ D : \neg r2 \end{array}$$

In the next step, LM is merged with N. Notice that B is present in the input constraints of N and will be added to the output constraints of the remaining output r.

$$\text{LM} := \begin{array}{l} q : t_0 \\ A : q \\ D : q \end{array} \longmapsto \begin{array}{l} r2 : t_0 \\ u : t_0 \\ A : u \wedge r2 \\ D : \neg r2 \end{array}$$

$$\text{N} := \begin{array}{l} r : t_2 \\ B : r \end{array} \longmapsto \begin{array}{l} v : t_2 \\ B : \neg v \end{array}$$

$$\text{LMN} := \begin{array}{l} q : t_0 \\ A : q \\ B : q \\ D : q \end{array} \longmapsto \begin{array}{l} u : t_0 \\ v : t_0 \\ A : u \wedge v \\ B : u \wedge \neg v \\ D : \neg(u \wedge v) \end{array}$$

Finally, LMN structure is joined with O on two ports, to produce the complete formula for the workflow.

$$\text{LMN} := \begin{array}{l} q : t_0 \\ A : q \\ B : q \\ D : q \end{array} \longmapsto \begin{array}{l} u : t_0 \\ v : t_0 \\ A : u \wedge v \\ B : u \wedge \neg v \\ D : \neg(u \wedge v) \end{array}$$

$$\text{O} := \begin{array}{l} u : t_3 \\ v : t_4 \\ C : u \wedge v \end{array} \longmapsto \begin{array}{l} w : t_3 t_4 \\ C : w \end{array}$$

$$\text{LMNO} := \begin{array}{l} q : t_0 \\ A : q \\ B : q \\ C : q \\ D : q \end{array} \longmapsto \begin{array}{l} w : t_0 \\ A : w \\ B : w \\ C : w \\ D : \neg w \end{array}$$

The previous examples have illustrated all three types of graph composition. The mechanism presented allows us to define polymorphic type formulas for entire workflows based on the type formulas of their constituent nodes and statically determine safeness of type composition, minimum requirements on input data and the polymorphic characteristics of the workflow.

For abstract nodes, this polymorphism can be used to define connectivity metadata, so that a component registry (such as myExperiment (Roure et al., 2007), or InforSense's Customer Hub [8] can classify the nodes not only by their input types, but also by the minimal properties required of that type, such as presence of certain attributes in the input. This issue is identified as one of the data-oriented abstractions involved in workflow and abstract node publishing in Giannadakis (2008).

3.3 Type formulas for Discovery Net nodes

The type formulas for the nodes of the Discovery Net relational subset are listed in Figure 10. Some of the formulas have been presented already in the discussion. They are listed again here for completeness. As before, r, s, u denote data variables on inputs and outputs.

In the rule for Group node, GA denotes the grouping attribute and AL to represent the aggregation function on the parameters (average, maximum, minimum, standard deviation...). For simplicity, a single grouping attribute and aggregation column is assumed.

4 Case study: Adverse drug reaction workflow

The polymorphic type framework presented in the previous section has been prototyped as a type inference engine, and here we present a worked example of how it is applied to a real-world scientific workflow implemented in Discovery Net relational subset. The engine establishes the workflow type profile, listing required inputs, reserved input names, guaranteed outputs and polymorphic parts of the inputs that are preserved in the output.

[8] www.inforsense.com/chub/

The case study is based around profiling a complex workflow with a large number of data transformation operations used in the context of studying adverse drug reactions. Its complexity presents a motivation for conducting design-time type checking analysis to avoid run-time data type errors. Furthermore, the workflow is designed in a top-down fashion in terms of reusable workflow fragments that conduct specific tasks. These fragments can be stored in a workflow repository to increase their re-usability in future applications, which also presents additional motivation for conducting type analysis.

4.1 Study description

All drugs in the UK undergo strictly controlled clinical trials before the release to ensure their general safety to the public. However, in these trials it is very difficult to establish potential harmful effects over a longer term, and only on certain patient groups, such as diabetes sufferers or patients with heart conditions. Late detection of adverse drug reactions increases the risks associated with drug development. Discovering a significant ADR may result in the drug being taken off the market, loss of reputation for the drug company and a waste of research effort. This increased risk also has the effect of multiplying drug costs, lowering the rate of innovation and negatively impacting the overall healthcare cost.

One promising approach to developing a more reliable methodology for detection of ADRs is based on primary care systems. UK has one of the most advanced primary care systems in the world in terms of computerization (Majeed, 2004), with databases containing longitudinal medical records and drug prescription data covering millions of people. Primary care data is available from a number of providers, such as General Practioners Researh Database (GPRD[9]), The Health Integration Network (THIN[10]) and MediPlus[11], containing full prescription data and observations for a large body of patients over an extended period of time (10-20 years).

The study presented in this section is a case-crossover design, where the rates of adverse events are observed in the same individuals while they are exposed and unexposed to the drug. There were two objectives to the study. First was to use the case–crossover method to identify how long after the introduction of a drug, can an ADR be identified. Second was to do this in a reusable fashion that can be adapted to different data sources and different conditions and reduce the analysis time from several weeks to a couple of days. The findings of the study were published in Molokhia et al. (2008).

The workflow in the analysis is concerned with deriving denominators (average time until event occurrence) for each year that the study covers. The abstract step schema is given in Figure 11, with six basic steps involved in the data flow:

[9] GPRD, www.gprd.com
[10] THIN, www.thin.com
[11] www.mediplus.com

$$\text{Delete(A)} := \begin{array}{l} r:t \\ A:r \end{array} \longmapsto \begin{array}{l} s:t \\ A:\neg s \end{array} \qquad \text{Select(A)} := \begin{array}{l} r:t \\ A:r \end{array} \longmapsto \begin{array}{l} s:\emptyset \\ A:s \end{array}$$

$$\text{Filter}(f_A) := \begin{array}{l} r:t \\ A:r \end{array} \longmapsto \begin{array}{l} s:t \\ A:s \end{array} \qquad \text{Group}(GA_A, AL_B) := \begin{array}{l} r:t \\ A:r \\ B:r \end{array} \longmapsto \begin{array}{l} s:\emptyset \\ A:s \\ B:s \end{array}$$

$$\text{Derive}(A, f_B) := \begin{array}{l} r:t \\ A:\neg r \\ B:r \end{array} \longmapsto \begin{array}{l} s:t \\ A:s \\ B:s \end{array} \qquad \text{Union} := \begin{array}{l} r:t \\ s:t \end{array} \longmapsto u:t$$

$$\text{Diff(A)} := \begin{array}{l} r:t_1 \\ s:t_2 \\ A:r \wedge s \end{array} \longmapsto \begin{array}{l} u:t_1 \\ A:u \end{array} \qquad \text{Join(A)} := \begin{array}{l} r:t_1 \\ s:t_2 \\ A:r \wedge s \end{array} \longmapsto \begin{array}{l} u:t_1 t_2 \\ A:u \end{array}$$

Figure 10: Type formulas for relational model nodes

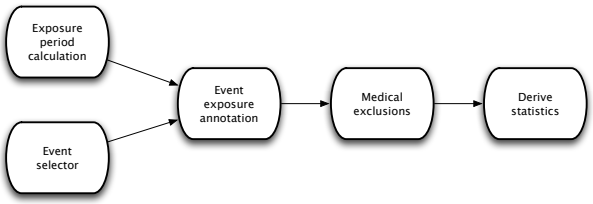

Figure 11: Abstract analysis data flow in the ADR study

1. **Exposure period calculation.** The prescription events are extracted matching a set of drugs that are being analysed – in our case statins and fibrates. Some basic data cleaning is performed to make sure that quantity is present in each case (0 quantity is used to denote regular prescription which is stopped for some reason). Exposure end date is defined as being 26 weeks after the start of prescription.

2. **Event selection.** Adverse myopathy events are extracted from the table of all medical events. Each event is assigned an identifier.

3. **Event exposure calculation.** Two data sets are joined, and it is observed for each event if it falls into one of the exposure periods. If so, it is marked as exposed, otherwise it is unexposed.

4. **Medical exclusions.** There are known medical conditions that cause myopathy, the patients suffering from those conditions are excluded from the event list.

5. **Drug exclusions.** There are also drugs known to cause myopathy as a side-effect of the treatment. Patients on those drugs are excluded from the event list.

6. **Derive statistics.** Denominators (time from start of exposure until the event) are calculated for each drug category, and Chi-square analysis is performed to determine the significance of exposed against unexposed events.

The concrete workflow in Figure 12 uses the relational subset of Discovery Net to implement the abstract design. The abstract steps are mapped to sets of nodes, with some inputs shared between the steps. The steps in the workflow are as follows:

1. Exposure period calculation
2. Event selection
3. Event exposure calculation
4. Medical exclusions
5. Drug exclusions
6. Derive statistics

In this section, the type properties of the workflow are investigated to determine what are its input data requirements and how does the workflow transform variable input parts in its output. The full polymorphic type formula for the workflow will be constructed using a step–wise procedure. The exposure period calculation and event exclusions will be considered first, and then the event joins and exposure definitions. The result will be a full description of the data transformation represented by the workflow.

4.2 Exposure period calculation

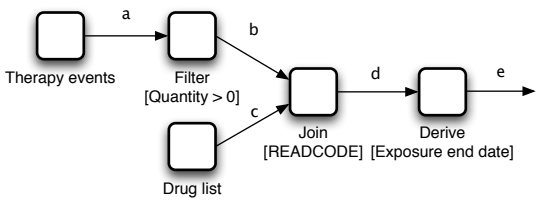

Figure 13: Exposure period calculation workflow

The derivation starts from the two data source nodes, one with the list of therapy events and the other with drugs that are considered in the study. The therapies are filtered

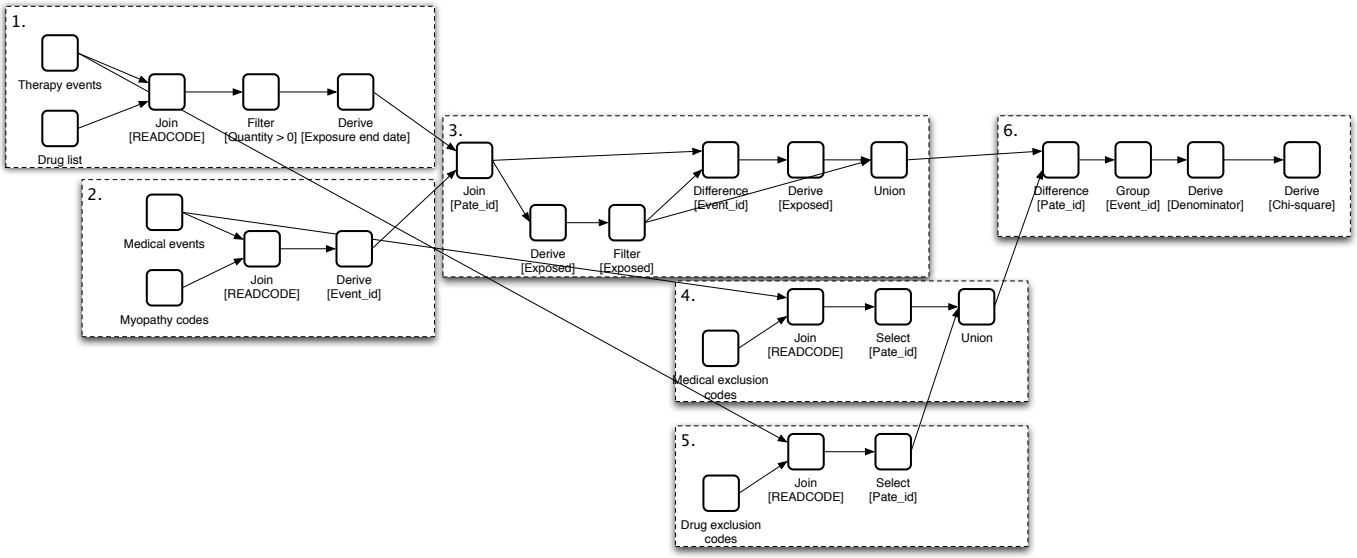

Figure 12: Analysis data flow annotated with segments

for the ones that have quantity specified, the two tables are joined, and the end date for exposure is specified, using the exposure start date attribute from the input.

Image in Figure 13 shows the workflow labelled with input and output variables. As a first step, the *Therapy events* and *Filter* are composed. Their types are resolved trivially, $t_1 = t_2$. Output conditions of the first and the input conditions of the second have no overlap in reserved variables, so their composition is also trivial. Finally, the input constraint of the *Filter* is mapped to the result input constraint.

$$\text{Therapy events} ::= \emptyset \longmapsto a : t_1$$

$$\text{Filter} ::= \begin{array}{l} a : t_2 \\ \text{Quantity} : a \end{array} \longmapsto \begin{array}{l} b : t_2 \\ \text{Quantity} : b \end{array}$$

$$\text{Result} ::= \begin{array}{l} a : t_1 \\ \text{Quantity} : a \end{array} \longmapsto \begin{array}{l} b : t_1 \\ \text{Quantity} : b \end{array}$$

In the next step, the result above is joined with the list of drugs, using the *Join* node.

$$\text{Filt. th.} ::= \begin{array}{l} a : t_1 \\ \text{Quantity} : a \end{array} \longmapsto \begin{array}{l} b : t_1 \\ \text{Quantity} : b \end{array}$$

$$\text{Drug list} ::= \emptyset \longmapsto c : t_3$$

$$\text{Join[Read.]} ::= \begin{array}{l} b : t_4 \\ c : t_5 \\ \text{Readcode} : b \wedge c \end{array} \longmapsto \begin{array}{l} d : t_4 t_5 \\ \text{Readcode} : d \end{array}$$

The type mappings are resolved and declarations are unified between the join inputs, and constraints resolved to produce:

$$\text{Result} ::= \begin{array}{l} a : t_1 \\ c : t_3 \\ \text{Readcode} : a \wedge c \\ \text{Quantity} : a \end{array} \longmapsto \begin{array}{l} d : t_1 t_3 \\ \text{Readcode} : d \\ \text{Quantity} : d \end{array}$$

Finally, this is sequentially composed with the *Derive* node, which creates the *Exposure-end-date*, based on the existing *Exposure-start-date* column.

$$\text{Derive} ::= \begin{array}{l} d : t_6 \\ \text{Exp-start-date} : d \\ \text{Exp-end-date} : \neg d \end{array} \longmapsto \begin{array}{l} e : t_6 \\ \text{Exp-start-date} : e \\ \text{Exp-end-date} : e \end{array}$$

Polymorphic type segments are resolved by assigning $t_6 = t_1 t_3$. Variable mapping of d is $vmap(d) = a \vee c$, and it holds that $cons(\text{Exposure-start-date}) = a \vee c$, and $cons(\text{Exposure-end-date}) = \neg(a \vee c)$, resulting in the full type formula for Exposure period calculation.

$$\text{Exposure period calculation} ::= \begin{array}{l} a : t_1 \\ c : t_3 \\ \text{Readcode} : a \wedge c \\ \text{Quantity} : a \\ \text{Exp-start-date} : a \vee c \\ \text{Exp-end-date} : \neg(a \vee c) \end{array} \longmapsto \begin{array}{l} e : t_1 t_3 \\ \text{Readcode} : e \\ \text{Quantity} : e \\ \text{Exp-start-date} : e \\ \text{Exp-end-date} : e \end{array}$$

4.3 Exclusion patients

The segment shown in Figure 14 shows the calculation of medical and drug exclusions. Despite the size, these are relatively simple to calculate. Join and select are combined as follows.

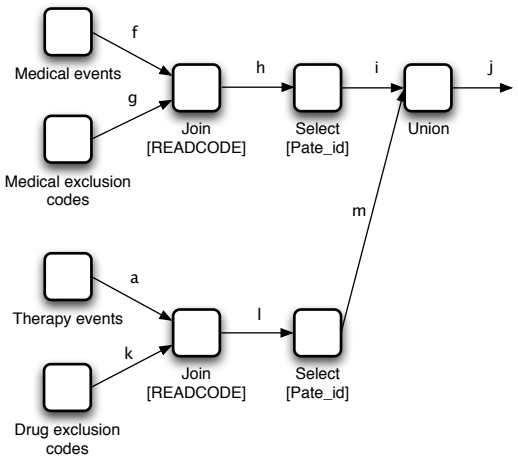

Figure 14: Exclusion computation workflow

$$\text{Join ::=} \begin{array}{l} f : t_7 \\ g : t_8 \\ \text{Readcode} : f \wedge g \end{array} \longmapsto \begin{array}{l} h : t_7 t_8 \\ \text{Readcode} : h \end{array}$$

$$\text{Select ::=} \begin{array}{l} h : t_9 \\ \text{Pate-id} : h \end{array} \longmapsto \begin{array}{l} i : \emptyset \\ \text{Pate-id} : i \end{array}$$

$$\text{Result ::=} \begin{array}{l} f : t_7 \\ g : t_8 \\ \text{Readcode} : f \wedge g \\ \text{Pate-id} : f \vee g \end{array} \longmapsto \begin{array}{l} i : \emptyset \\ \text{Pate-id} : i \end{array}$$

Identical procedure is repeated for the lower part of the workflow, producing:

$$\text{Result ::=} \begin{array}{l} a : t_1 \\ k : t_{10} \\ \text{Readcode} : a \wedge k \\ \text{Pate-id} : a \vee k \end{array} \longmapsto \begin{array}{l} m : \emptyset \\ \text{Pate-id} : m \end{array}$$

The union node only conjuncts the constraints, giving the final type formula for Exclusions segment:

$$\text{Exclusions ::=} \begin{array}{l} f : t_7 \\ g : t_8 \\ a : t_1 \\ k : t_{10} \\ \text{Readcode} : (f \wedge g) \wedge (a \wedge k) \\ \text{Pate-id} : (a \vee k) \wedge (f \vee g) \end{array} \longmapsto \begin{array}{l} m : \emptyset \\ \text{Pate-id} : m \end{array}$$

4.4 Event occurrences

The segment in Figure 15 calculates the myopathy events. It takes in the list of all medical events, excluding certain patients, and joining with the myopathy codes specified. Result is the list of all events observed, and a helper column with event identifier is generated.

The difference formula takes in the previous type formula from Exclusions, and the polymorphic segment of the input is mapped from *Medical events*. This result is

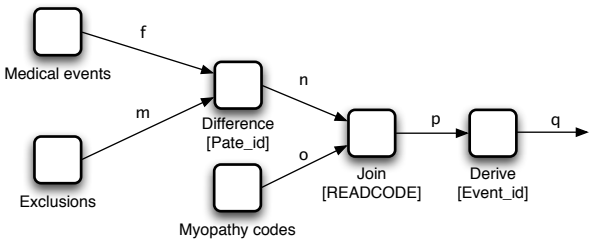

Figure 15: Event occurrences workflow

then composed with the myopathy event list using *Join*.

$$\text{Filtered events ::=} \begin{array}{l} f : t_7 \\ g : t_8 \\ a : t_1 \\ k : t_{10} \\ \text{Readcode} : (f \wedge g) \wedge (a \wedge k) \\ \text{Pate-id} : (a \vee k) \wedge (f \vee g) \end{array} \longmapsto \begin{array}{l} n : t_7 \\ \text{Pate-id} : n \end{array}$$

$$\text{Myopathy codes::=} \emptyset \longmapsto o : t_{11}$$

$$\text{Joined result ::=} \begin{array}{l} f : t_7 \\ g : t_8 \\ a : t_1 \\ k : t_{10} \\ o : t_{11} \\ \text{Readcode} : (f \wedge g) \wedge (a \wedge k) \wedge (f \wedge o) \\ \text{Pate-id} : (a \vee k) \wedge (f \vee g) \end{array} \longmapsto \begin{array}{l} p : t_7 t_{11} \\ \text{Readcode} : p \\ \text{Pate-id} : p \end{array}$$

This result is then combined with *Derive* to produce the final type formula for the Event occurences segment. With type mapping $tmap(t_{12}) = t_7 t_{11}$, and variable mapping $vmap(p) = f \vee o$.

$$\text{Derive ::=} \begin{array}{l} p : t_{12} \\ \text{Event-id} : \neg p \end{array} \longmapsto \begin{array}{l} q : t_{12} \\ \text{Event-id} : q \end{array}$$

$$\text{Event calc. ::=} \begin{array}{l} f : t_7 \\ g : t_8 \\ a : t_1 \\ k : t_{10} \\ o : t_{11} \\ \text{Readcode} : (f \wedge g) \wedge (a \wedge k) \wedge (f \wedge o) \\ \text{Pate-id} : (a \vee k) \wedge (f \vee g) \\ \text{Event-id} : \neg(f \vee o) \end{array} \longmapsto \begin{array}{l} q : t_7 t_{11} \\ \text{Readcode} : q \\ \text{Pate-id} : q \\ \text{Event-id} : q \end{array}$$

4.5 Deriving event exposure (1)

In order to derive event exposure, as shown in Figure 16, we first perform the join of prescription periods and calculated

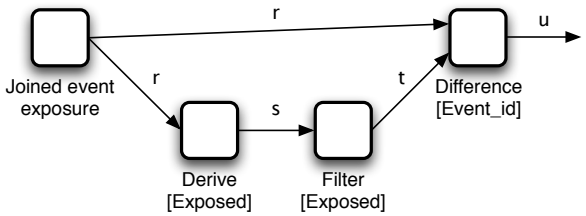

Figure 16: Deriving event exposure - first part

events, based on the identifier of the patient. The resulting formula is:

$$\begin{array}{rl} & a : t_1 \\ & c : t_3 \\ & f : t_7 \\ & g : t_8 \\ & k : t_{10} \\ & o : t_{11} \\ \text{Joined event exp.} ::= & \text{Readcode} : (a \wedge c) \wedge (f \wedge g) \wedge \\ & (a \wedge k) \wedge (f \wedge o) \\ & \text{Quantity} : a \\ & \text{Exp-start-date} : a \vee c \\ & \text{Exp-end-date} : \neg(a \vee c) \\ & \text{Pate-id} : (a \vee k) \wedge (f \vee g) \\ & \text{Event-id} : \neg(f \vee o) \\ & \\ & r : t_1 t_3 t_7 t_{11} \\ & \text{Readcode} : r \\ & \text{Pate-id} : r \\ \longmapsto & \text{Event-id} : r \\ & \text{Quantity} : r \\ & \text{Exp-start-date} : r \\ & \text{Exp-end-date} : r \end{array}$$

The lower branch of the workflow composes this formula with a *Derive* and a *Filter*. Exposed column in *Derive* requires the presence of event date, exposure start date and exposure end date, as shown in the node formula.

$$\begin{array}{rl} & r : t_{13} \\ & \text{Exp-start-date} : r \\ \text{Derive[Exposed]} ::= & \text{Exp-end-date} : r \\ & \text{Event-date} : r \\ & \text{Exposed} : \neg r \\ & \\ & s : t_{13} \\ & \text{Exp-start-date} : s \\ \longmapsto & \text{Exp-end-date} : s \\ & \text{Event-date} : s \\ & \text{Exposed} : s \end{array}$$

The resulting formula introduces the event date constraint into the context on the left hand side. Note that, even though it is obvious that event date needs to be present in the *Medical events* table, and type t_7, the typing algorithm does not know that, and it merely requires that any of the inputs constructing the type $t_1 t_3 t_7 t_{11}$ has it present.

$$\begin{array}{rl} & a : t_1 \\ & c : t_3 \\ & f : t_7 \\ & g : t_8 \\ & k : t_{10} \\ & o : t_{11} \\ \text{Exp. events derived} ::= & \text{Readcode} : (a \wedge c) \wedge (f \wedge g) \wedge \\ & (a \wedge k) \wedge (f \wedge o) \\ & \text{Quantity} : a \\ & \text{Exp-start-date} : a \vee c \\ & \text{Exp-end-date} : \neg(a \vee c) \\ & \text{Pate-id} : (a \vee k) \wedge (f \vee g) \\ & \text{Event-id} : \neg(f \vee o) \\ & \text{Event-date} : a \vee c \vee f \vee o \\ & \text{Exposed} : \neg(a \vee c \vee f \vee o) \\ & \\ & s : t_1 t_3 t_7 t_{11} \\ & \text{Readcode} : s \\ & \text{Pate-id} : s \\ & \text{Event-id} : s \\ \longmapsto & \text{Quantity} : s \\ & \text{Exp-start-date} : s \\ & \text{Exp-end-date} : s \\ & \text{Event-date} : s \\ & \text{Exposed} : s \end{array}$$

Filter node merely performs the test on the Exposed attribute, which is guaranteed to be present by the previous node. The result, a list of exposed events, is fed as a subtract into the *Difference* node which preserves the type of the first input, and produces the list of unexposed events. Note that the preconditions of the lower branch remain, however the output constraints of the subtract are not preserved by the difference. The full formula for the workflow up to this point is:

$$\begin{array}{rl} & a : t_1 \\ & c : t_3 \\ & f : t_7 \\ & g : t_8 \\ & k : t_{10} \\ & o : t_{11} \\ & \text{Readcode} : (a \wedge c) \wedge (f \wedge g) \wedge \\ \text{Unexposed events} ::= & (a \wedge k) \wedge (f \wedge o) \\ & \text{Quantity} : a \\ & \text{Exp-start-date} : a \vee c \\ & \text{Exp-end-date} : \neg(a \vee c) \\ & \text{Pate-id} : (a \vee k) \wedge (f \vee g) \\ & \text{Event-id} : \neg(f \vee o) \\ & \text{Event-date} : a \vee c \vee f \vee o \\ & \text{Exposed} : \neg(a \vee c \vee f \vee o) \\ & \\ & u : t_1 t_3 t_7 t_{11} \\ & \text{Readcode} : u \\ & \text{Pate-id} : u \\ \longmapsto & \text{Event-id} : u \\ & \text{Quantity} : u \\ & \text{Exp-start-date} : u \\ & \text{Exp-end-date} : u \end{array}$$

4.6 Deriving event exposure (2)

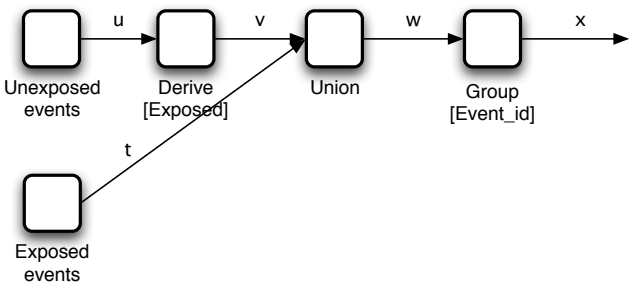

Figure 17: Deriving event exposure - second part

In this segment, the derivation from previous step is repeated, only this time on the list of unexposed events, and then unioned with the exposed events. Grouping node preserves the first value encountered, and serves only to eliminate events mapped to multiple periods, changing nothing in the type formula. Therefore the full event exposure formula becomes:

$$
\begin{array}{rl}
 & a : t_1 \\
 & c : t_3 \\
 & f : t_7 \\
 & g : t_8 \\
 & k : t_{10} \\
 & o : t_{11} \\
 & \text{Readcode} : (a \wedge c) \wedge (f \wedge g) \wedge \\
\text{Annotated events} ::= & (a \wedge k) \wedge (f \wedge o) \\
 & \text{Quantity} : a \\
 & \text{Exp-start-date} : a \vee c \\
 & \text{Exp-end-date} : \neg(a \vee c) \\
 & \text{Pate-id} : (a \vee k) \wedge (f \vee g) \\
 & \text{Event-id} : \neg(f \vee o) \\
 & \text{Event-date} : a \vee c \vee f \vee o \\
 & \text{Exposed} : \neg(a \vee c \vee f \vee o) \\
 & \\
 & x : t_1 t_3 t_7 t_{11} \\
 & \text{Readcode} : x \\
 & \text{Pate-id} : x \\
 & \text{Event-id} : x \\
\longmapsto & \text{Quantity} : x \\
 & \text{Exp-start-date} : x \\
 & \text{Exp-end-date} : x \\
 & \text{Event-date} : x \\
 & \text{Exposed} : x
\end{array}
$$

4.7 Calculate statistics

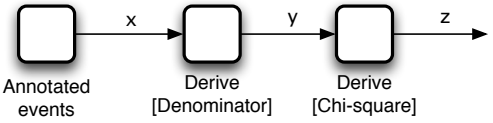

Figure 18: Workflow for calculating statistics

The last step in the workflow is the calculation of denominators and chi–square statistic, Figure 18. Denominators are calculated by finding out the time, for each event, and use the event date and exposure start date. Chi-square is implemented by an external service, and requires the denominators and the drug code to be present, so the drug code is added to the list of input constraints. Therefore, the full type formula for the entire workflow is:

$$
\begin{array}{rl}
 & a : t_1 \\
 & c : t_3 \\
 & f : t_7 \\
 & g : t_8 \\
 & k : t_{10} \\
 & o : t_{11} \\
 & \text{Readcode} : (a \wedge c) \wedge (f \wedge g) \wedge \\
 & (a \wedge k) \wedge (f \wedge o) \\
\text{Analysis} ::= & \text{Quantity} : a \\
 & \text{Exp-start-date} : a \vee c \\
 & \text{Exp-end-date} : \neg(a \vee c) \\
 & \text{Pate-id} : (a \vee k) \wedge (f \vee g) \\
 & \text{Event-id} : \neg(f \vee o) \\
 & \text{Event-date} : a \vee c \vee f \vee o \\
 & \text{Drug-code} : a \vee c \vee f \vee o \\
 & \text{Exposed} : \neg(a \vee c \vee f \vee o) \\
 & \text{Denominator} : \neg(a \vee c \vee f \vee o) \\
 & \text{Chi-square} : \neg(a \vee c \vee f \vee o) \\
 & \\
 & z : t_1 t_3 t_7 t_{11} \\
 & \text{Readcode} : z \\
 & \text{Pate-id} : z \\
 & \text{Event-id} : z \\
 & \text{Quantity} : z \\
\longmapsto & \text{Exp-start-date} : z \\
 & \text{Exp-end-date} : z \\
 & \text{Event-date} : z \\
 & \text{Drug-code} : z \\
 & \text{Exposed} : z \\
 & \text{Denominator} : z \\
 & \text{Chi-square} : z
\end{array}
$$

The output depends on six polymorphic variables, four of which are passed into the output. Therefore, adding extra columns to t_8 and t_{10} will not affect the result. Eleven variable constraints are present on the inputs, while eleven variables are guaranteed in the output.

5 Summary and conclusions

This paper introduced a polymorphic type framework for scientific workflow systems that support composable data attributes (such as the relational model and XML), and provided a mechanism for assembling a workflow's polymorphic type signature based on the type formulas of individual nodes. Such signature specifies the minimum set of attributes the workflow requires from the data input and maps the variable part of the output to the variable part of the input, thus providing an implementation independent type signature suitable for use in workflow registries.

The type formalism was demonstrated by presenting the node formulas of a generic relational workflow system and applying the type inference engine prototype to profiling a medical informatics study implemented in Discovery Net.

The argument for omitting typing from workflow systems is that in the open world (Goble, 2004) in which newly discovered services are dynamically selected and plugged into workflows, there is little point in profiling their exact inputs and outputs, and these need to be resolved at run-time through automatic or semi-automatic resolution, also known as shimming (Hull et al., 2004; Ambite and Kappor, 2007). This is opposed to the closed world view, in which the set of workflow components is largely known in advance, or at least the interfaces they present to the world are. In the latter case, which covers the data analytics systems listed above, typing is not only possible but highly desirable.

Adding polymorphic typing improves a workflow system in two ways. First, during node composition, each node input can declare all its type requirements in advance, thereby reducing the effort needed in exploring individual type requirements one by one. Second, with typed systems that do not employ polymorphism, workflows determine the output data type given input data types, but do not make a distinction between outputs that are guaranteed by the type requirements of the inner workflow, and the ones that originate from the current input data set, so the workflow does not know which parts of its data output can be made variable if some inputs are abstracted. This restriction is lifted with the introduction of polymorphism.

These improvements directly facilitate efforts in guided construction of workflows and workflow reuse (Wroe et al., 2007), and also improve the usability of node registries by allowing type composition of nodes that have not yet been retrieved from the registry, based purely on their specifications. Finally, design of abstract workflows is greatly enhanced by providing input and output constraints on the inputs and outputs of context holes that are to be filled by concrete node implementations.

The formalism introduced is not restricted only to strongly typed workflow systems. The typing algorithms require information about the type transformation of each node and once these are present, they generate a full type checking system for workflows in the system. For example, Taverna workflow system has no static type checking, since it operates with XML documents which are only retrieved at runtime. However, its nodes can define their type requirements, in terms of schema attributes required, which can then be used to infer the polymorphic type formula for the entire workflow.

5.1 Alternative approaches

The approach taken in this paper was to develop flexible formalisms that are immediately applicable to a wide range of scientific workflow systems, with a polymorphic typing framework used to model propagation of types through the graph and represent type properties of a graph segment. Another possible approach to representing node behaviour is to expand the π-calculus formalism used to model execution semantics in Curcin et al. (2009) to include typing constructs (Milner, 1993; Pierce and Sangiorgi, 1996; Yoshida and Hennessy, 1999) to characterise the composition of type transformations. This has been investigated previously outside of the workflow context, and used to derive encodings of several λ calculi using π-calculus (Milner, 1992; Turner, 1996), by associating types with the names in the calculus and separating types of values from types of links used to send those values, as demonstrated in Base-π and simply typed π-calculus examples presented in Sangiorgi and Walker (2001).

5.2 Future work

The data collections considered for typing have been restricted to type structures with well-defined attributes, however extending support to the full set of standard data structures, such as lists, trees and maps, would increase its applicability. The list semantics have been investigated within the context of Taverna using category theory in Turi et al. (2007). The map semantics, which can model indexing mechanisms in the data sets, would be especially relevant to future developments in workflow-based data mining systems that need to efficiently perform operations on large data sets, since it would allow the indexing information to be incorporated into node type information and support processing decisions based on it.

REFERENCES

Altintas, I., Berkley, C., Jaeger, E., Jones, M., Ludascher, B., and Mock, S. (2004). Kepler: an extensible system for design and execution of scientific workflows. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management, 2004*, pages 423–424.

Ambite, J. L. and Kappor, D. (2007). Automatically composing data workflows with relational descriptions and shim services. In Aberer, K., Choi, K.-S., Noy, N., Allemang, D., Lee, K.-I., Nixon, L. J. B., Golbeck, J., Mika, P., Maynard, D., Schreiber, G., and Cudr-Mauroux, P., editors, *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Semantic Web Conference (ISWC/ASWC2007), Busan, South Korea*, volume 4825 of *LNCS*, pages 15–28, Berlin, Heidelberg. Springer Verlag.

Berthold, M. R., Cebron, N., Dill, F., Gabriel, T. R., Kötter, T., Meinl, T., Ohl, P., Sieb, C., Thiel, K., and Wiswedel, B. (2007). Knime: The konstanz information miner. In *Studies in Classification, Data Analysis, and Knowledge Organization (GfKL 2007)*. Springer.

Chen, J. and van der Aalst, W. M. P. (2007). On scientific workflows. *IEEE Computer Society's Technical Committee for Scalable Computing*, 9.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of ACM*, 13(6):377–387.

Curcin, V. (2009). *Workflow Modelling of Scientific Processes*. PhD thesis, Imperial College London.

Curcin, V., Ghanem, M., and Guo, Y. (2009). Analysing workflows with Computational Tree Logic. *Special edition on Recent Advances in e-Science of Journal of Cluster Computing (to appear)*.

Curcin, V., Ghanem, M., Wendel, P., and Guo, Y. (2007). Heterogeneous workflows in scientific workflow systems. In Shi, Y., van Albada, G. D., Dongarra, J., and Sloot, P. M. A., editors, *International Conference on Computational Science (3)*, volume 4489 of *Lecture Notes in Computer Science*, pages 204–211. Springer.

Date, C. J. (1995). *An Introduction to Database Systems, 6th Edition*. Addison-Wesley.

Demšar, J., Zupan, B., Leban, G., and Curk, T. (2004). Orange: from experimental machine learning to interactive data mining. In *PKDD '04: Proceedings of the 8th European Conference on Principles and Practice of Knowledge Discovery in Databases*, pages 537–539, New York, NY, USA. Springer-Verlag New York, Inc.

Fox, G., Chiu, K., and Buyya, R., editors (2007). *Third International Conference on e-Science and Grid Computing, e-Science 2007, 10-13 December 2007, Bangalore, India*. IEEE Computer Society.

Ghanem, M., Curcin, V., Wendel, P., and Guo, Y. (2008). Building and using analytical workflows in Discovery Net. In Dubitzky, W., editor, *Data mining on the Grid*, pages 119–140. John Wiley and Sons.

Giannadakis, N. (2008). *Enabling Cross-domain Workflow Reuse*. PhD thesis, Imperial College London.

Gil, Y., Deelman, E., Ellisman, M., Fahringer, T., Fox, G., Gannon, D., Goble, C., Livny, M., Moreau, L., and Myers, J. (2007). Examining the challenges of scientific workflows. *Computer*, 40(12):24–32.

Goble, C. (2004). Building ad hoc (personal) workflows in an open world: myGrid experiences. In *Proceedings of the 12th Global Grid Forum Workshop (Brussels, Belgium, Sept. 2004)*.

Hey, A. and Trefethen, A. E. (2002). The UK e-Science core programme and the Grid. *Future Generation of Computer Systems*, 18(8):1017–1031.

Hull, D., Stevens, R., Lord, P., and Goble, C. (2004). Integrating bioinformatics resources using shims. In *ISMB/ECCB (Supplement of Bioinformatics)*.

Hull, D., Wolstencroft, K., Stevens, R., Goble, C., Pocock, M. R., Li, P., and Oinn, T. (2006). Taverna: A tool for building and running workflows of services. *Nucleic Acids Research*, 34:W729–W732. Web Server Issue.

Ludäscher, B. and Goble, C. (2005). Introduction to the special section on scientific workflows. *SIGMOD Records*, 34(3):3–4.

Ludäscher, B. and Raschid, L., editors (2005). *Data Integration in the Life Sciences, Second International-Workshop, DILS 2005, San Diego, CA, USA, July 20-22, 2005, Proceedings*, volume 3615 of *Lecture Notes in Computer Science*. Springer.

MacQueen, D. B., Plotkin, G. D., and Sethi, R. (1984). An ideal model for recursive polymorphic types. In *POPL*, pages 165–174.

Mahoui, M., Lu, L., Gao, N., Li, N., Chen, J., Bukhres, O., and Miled, Z. B. (2005). A dynamic workflow approach for the integration of bioinformatics services. *Cluster Computing*, 8(4):279–291.

Majeed, A. (2004). Sources, uses, strengths and limitations of data collected in primary care in england. *Health Statistics Quarterly*, 21:5–14.

Milner, R. (1978). A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375.

Milner, R. (1992). Functions as processes. *Journal of Mathematical Structures in Computer Science*, 2(2):119–141.

Milner, R. (1993). The polyadic pi-calculus: a tutorial. In Bauer, F. L., Brauer, W., and Schwichtenberg, H., editors, *Logic and Algebra of Specification*, pages 203–246. Springer-Verlag.

Molokhia, M., McKeigue, P., Curcin, V., and Majeed, A. (2008). Statin induced myopathy and myalgia: Time trend analysis and comparison of risk associated with statin class from 1991-2006. *PLoS ONE*, 3(6):e2522.

Pierce, B. C. and Sangiorgi, D. (1996). Typing and subtyping for mobile processes. *Journal of Mathematical Structures in Computer Science*, 6(5):409–454.

Roure, D. D., Goble, C. A., and Stevens, R. (2007). Designing the myexperiment virtual research environment for the social sharing of workflows. In Fox et al. (2007), pages 603–610.

Sangiorgi, D. and Walker, D. (2001). *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press.

Taylor, I., Shields, M., Wang, I., and Harrison, A. (2005). Visual Grid Workflow in Triana. *Journal of Grid Computing*, 3(3-4):153–169.

Taylor, I. J., Deelman, E., and Gannon, D. B. (2006). *Workflows for e-Science: Scientific Workflows for Grids*. Springer.

Turi, D., Missier, P., Goble, C. A., Roure, D. D., and Oinn, T. (2007). Taverna workflows: Syntax and semantics. In Fox et al. (2007), pages 441–448.

Turner, D. N. (1996). *The Polymorphic Pi-Calculus: Theory and Implementation*. PhD thesis, Department of Computer Science, University of Edinburgh.

van den Bussche, J. and Waller, E. (1999). Type inference in the polymorphic relational algebra. In *PODS '99: Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 80–90, New York, NY, USA. ACM.

Wroe, C., Goble, C., Goderis, A., Lord, P., Miles, S., Papay, J., Alper, P., and Moreau, L. (2007). Recycling workflows and services through discovery and reuse: Research articles. *Concurrency and Computation : Practice and Experience*, 19(2):181–194.

Xu, F., Eres, H., Tao, F., and Cox, S. (2004). Workflow support for advanced Grid-enabled computing. In *Proceedings of UK e-Science AHM 2004 Conference*. http://eprints.ecs.soton.ac.uk/9727/ Last accessed May 2008.

Yoshida, N. and Hennessy, M. (1999). Subtyping and locality in distributed higher order processes (extended abstract). In Baeten, J. C. M. and Mauw, S., editors, *CONCUR '99: Concurrency Theory (10th International Conference, Eindhoven, The Netherlands)*, volume 1664 of *lncs*, pages 557–572. Springer Verlag.