Typetheoretic Approach to the Shimming Problem in Scientific Workflows
(Technical Report TR-BIGDATA-102013-KLC)

Andrey Kashlev, Shiyong Lu, Artem Chebotko
Wayne State University
Department of Computer Science
5057 Woodward Ave, Detroit, Michigan, 48202, USA
andrey.kashlev@wayne.edu

Abstract
When composing Web services into scientific workflows, users often face the so-called shimming problem when connecting two related but incompatible components. The problem is addressed by inserting a special kind of adaptors, called shims, that perform appropriate data transformations to resolve data type inconsistencies. However, existing shimming techniques provide limited automation and burden users with having to define ontological mappings, generate data transformations, and even manually write shimming code. In addition, these approaches insert many visible shims that clutter workflow design and distract user’s attention from functional components of the workflow. To address these issues, we 1) reduce the shimming problem to a runtime coercion problem in the theory of type systems, 2) propose a scientific workflow model and define the notion of well-typed workflows, 3) develop an algorithm to typecheck workflows, 4) design a function that inserts “invisible shims”, or runtime coercions into workflows, thereby solving the shimming problem for any well-typed workflow, 5) implement our automated shimming technique, including all the proposed algorithms, lambda calculus, type system, and translation function in our VIEW system and present two case studies to validate our approach.

1 Introduction
Web Service composition plays a key role in the fields of services computing [1, 2, 3, 4, 5] and scientific workflows [6, 7, 8, 9]. Oftentimes composing autonomous third-party Web services into workflows requires using intermediate components, called shims, to mediate syntactic and semantic incompatibilities between different heterogeneous components.

Consider a workflow \( W_s \) in Fig. 1 comprised of two Web services – \( WS_1 \) and \( WS_2 \). \( WS_2 \) expects an XML document that differs from that returned by \( WS_1 \). Particularly, \( WS_2 \) expects an XML document with three child elements, rather than four, and the \( concentr \) element should be of type Double rather than Float. Besides, \( concentr \) element should be
the last element under data rather than the second one. To resolve this incompatibility (shown as a dashed line in Fig. 1) and ensure successful workflow execution, we need to obtain and insert the shim that will perform appropriate data transformation.

Determining where the shim is needed, obtaining appropriate shim and inserting it is known as the shimming problem, whose significance is widely recognized by the Web Service community [10, 11, 12, 13, 14, 15]. Existing approaches to the shimming problem have the following limitations.

First, existing techniques are not automated and burden users by requiring them to generate transformation scripts, define mappings to and from domain ontologies, and even write shimming code [13, 16, 17]. We believe these requirements are difficult and make workflow design counterproductive for non-technical users.
Second, current approaches produce cluttered workflows with many visible shims that distract users from main workflow components that perform useful work. Furthermore, recent workflow studies [18, 19] show that the percentage of shim components in workflows registered in myExperiment portal (www.myexperiment.org) has grown from 30% in 2009 [18] to 38% in 2012 [19]. These numbers indicate that such explicit shimming tends to make workflows even messier overtime, which further diminishes the usefulness of these techniques.

Third, many shimming techniques only apply under a particular set of circumstances that are hard to guarantee or even predict. Some approaches (e.g., [13, 16, 20, 21]) apply only when all the right shims are supplied by Web Service providers and are properly annotated beforehand, and/or when required shims can be generated by automated agents (e.g., XQuery-based shims [21]), which cannot be guaranteed for any practical class of workflows. Such uncertainty makes these techniques unreliable in the eyes of end users (domain scientists) who need assurance that their workflows will run.

Finally, while these efforts focus on resolving structural differences between complex types of Web services [13, 16, 20], they cannot mediate primitive types, such as \texttt{Int} or \texttt{Double}.

To address these issues, we propose a fully automated technique that, given a workflow creates and inserts suitable shims.Inserted shims transform data appropriately allowing successful workflow execution. Specifically, we

1. reduce the shimming problem to a runtime coercion problem in the theory of type systems,
2. propose a scientific workflow model and define the notion of well-typed workflows,
3. develop an algorithm to translate workflows into equivalent lambda expressions,
4. develop an algorithm to typecheck workflow expressions,
5. design a function that inserts “invisible shims” (coercions) into workflows, thereby solving the shimming problem for any well-typed workflow,
6. implement our automated shimming technique and present two case studies to validate the proposed approach to mediate Web services.

To our best knowledge, this work is the first one to reduce the shimming problem to the coercion problem and to propose a fully automated solution with no human involvement. Moreover, our technique frees workflow design from visible shims by dynamically inserting transparent coercions in workflows during execution time (implicit shimming). The proposed solution automatically mediates both structural data types, such as complex types of Web service inputs/outputs as well as primitive data types, such as \texttt{Int} and \texttt{Double}.

2 **Scientific Workflow Model**

Scientific workflows consist of one or more computational components connected to each other and possibly to some input data products. Each of these components can be viewed as a black box with well defined input and output ports. Every component is itself another workflow, either primitive or composite. Primitive workflows are bound to
executable components, such as Web services, scripts, or high performance computing (HPC) services and can be viewed as atomic entities. Composite workflows consist of multiple building blocks connected to one another via data channels. Each of these building blocks can be either a workflow or a data product. In the following we formalize the scientific workflow model used in this paper.

**Definition 2.1 (Port).** A port is a pair \((id, type)\) consisting of a unique identifier and a data type associated with this port. We denote input and output ports as \(ip_i: T_i\) and \(op_j: T_j\), respectively, where \(ip_i\) and \(op_j\) are identifiers, and \(T_i\) and \(T_j\) are port types.

**Definition 2.2 (Data Product).** A data product is a triple \((id, value, type)\) consisting of a unique identifier, a value and a type associated with this data product. We denote each data product as \(dp_i: T_i\), where \(dp_i\) is the identifier, and \(T_i\) is the type of the data product.

Given a workflow \(W\) and the set of its constituent workflows \(W^*\), we use \(W.p_j\) to denote port \(p_j\) of \(W\) (be it input or output port) and \(W.W^*.IP (W.W^*.OP)\) to represent the union of sets of input (output) ports of all constituent workflows of \(W\). Whenever it is clear from the context we omit the leading “\(W\)”. Formally,

\[
W^*.IP = \{ip_i \mid ip_i \in W_i.IP, W_i \in W^*\}
\]

\[
W^*.OP = \{op_k \mid op_k \in W_i.OP, W_i \in W^*\}
\]

**Definition 2.3 (Scientific workflow).** A scientific workflow \(W\) is a 9-tuple \((id, IP, OP, W^*, DP, DC_{in}, DC_{out}, DC_{mid}, DC_{idp})\), where

1. \(id\) is a unique identifier,
2. \(IP = \{ip_0, ip_1, ..., ip_n\}\) is an ordered set of input ports,
3. \(OP = \{op_0, op_1, ..., op_m\}\) is an ordered set of output ports,
4. \(W^* = \{W_0, W_1, ..., W_r\}\) is a set of constituent workflows used in \(W\). Each \(W_i \in W^*\) is another 9-tuple,
5. \(DP = \{dp_0, dp_1, ..., dp_p\}\) is a set of data products,
6. \(DC_{in} : IP \rightarrow W^*.IP\) is an inverse-functional one-to-many mapping. \(DC_{in}\) is a set of ordered pairs:

\[
DC_{in} \subseteq \{(ip_i, ip_k) \mid ip_i \in IP, ip_k \in W_i.IP, W_i \in W^*\}
\]

That is, each pair in \(DC_{in}\) represents a data channel connecting input port \(ip_i\) to an input port \(ip_k\) of some component \(W_i \in W^*\).

7. \(DC_{out} : W^*.OP \rightarrow OP\) is an inverse-functional one-to-many mapping. \(DC_{out}\) is a set of ordered pairs:

\[
DC_{out} \subseteq \{(op_j, op_k) \mid op_j \in W_i.OP, W_i \in W^*, op_k \in OP\}
\]

That is, each pair in \(DC_{out}\) represents a data channel connecting output port \(op_j\) of some component \(W_i \in W^*\) to an output port \(op_k\) of \(OP\).

8. \(DC_{mid} : W^*.OP \rightarrow W^*.IP\) is an inverse-functional one-to-many mapping. \(DC_{mid}\) is a set of ordered pairs:

\[
DC_{mid} \subseteq \{(op_j, ip_k) \mid op_j \in W_i.OP, ip_k \in W_m.IP, W_i, W_m \in W^*\}
\]

That is, each pair in \(DC_{mid}\) represents a data channel connecting an output port \(op_j\) of some component \(W_i \in W^*\) with an input port \(ip_k\) of another component \(W_m \in W^*\).

9. \(DC_{idp} : DP \rightarrow W^*.IP\) is an inverse-functional one-to-many mapping. \(DC_{idp}\) is a set of
ordered pairs:

$$DC_{idp} \subseteq \{(dp_i, ip_k) \mid dp_i \in DP, ip_k \in W_j . IP, W_j \in W^*\}.$$ 

That is, each pair in $DC_{idp}$ represents a data channel that connects a data product $dp_i \in DP$ to the input port $ip_k$ of some component $W_j \in W^*$. 

Fig. 2 shows seven workflows that we will reference in this paper as $W_a, W_b, W_c, W_d, W_e, W_f,$ and $W_g$ respectively. These seven workflows use other workflows as their building blocks. Such constituent workflows are shown as blue boxes with their ids written inside each box. Ports appear as red pins pointing right (input) or left (output). Finally, data products are visualized as yellow boxes with their values placed inside (e.g., “true” in $W_e$ in Fig. 2).

Because the order of input arguments of a workflow matters (e.g., $Divide$ workflow in $W_f$ in Fig. 2), we use ordered set $IP$ to store a list of input ports. We use the term data channel to refer to a wire, connecting a workflow port to a data product or another port. All entries from the set $\{DC_{in} \cup DC_{mid} \cup DC_{out} \cup DC_{idp}\}$ are data channels.

As we show in later sections, a workflow is represented as a lambda expression. To simplify lambda expressions, we focus on workflows with a single output port. We are currently extending our approach to allow set $OP$ with a cardinality greater than one. Our definition requires that every workflow and every data product has a unique id. For simplicity we also require that for any workflow $W$, all ports of $W$ and all ports of all workflows in $W^*$ have unique ids.

We model workflow $W_d$ in Fig. 2 as a 9-tuple, where $id = "W_d", IP = \emptyset, OP = \{(op_9, Float)\}, W^* = \{Mean, Sqrt\}, DP = \{(dp_0, 3, Int), (dp_1, 5, Int), (dp_2, 4, Int)\}, DC_{in} = \emptyset, DC_{out} = \{((Sqrt, op_5), op_3), (dp_1, (Mean, ip_4))\}, (dp_2, (Mean, ip_5)).$ Workflow $W_e$, on the other hand does not have concrete input data products connected to its inputs. We model it using 9-tuple with $id = "W_e", IP = \{(ip_0, Int), (ip_1, Int), (ip_2, Int)\}, OP = \{(op_9, Double)\}, W^* = \{Mean, Sqrt\}, DP = \emptyset, DC_{in} = \{(ip_0, (Mean, ip_3))\}, (ip_1, (Mean, ip_4)), (ip_2, (Mean, ip_5)), DC_{out} = \{((Sqrt, op_5), op_3), (dp_1, (Mean, ip_4))\}, DC_{idp} = \emptyset.$

**Definition 2.4 (Primitive workflow).** A workflow $W$ is primitive if and only if it has both input ports and output ports, and $W$ has neither constituent components, nor data
products, nor data channels. Formally, $W$ is primitive iff
\[ W.IP \neq \emptyset \land W.OP \neq \emptyset \land W.W^* = W.DP = W.DC_{in} = W.DC_{out} = W.DC_{mid} = W.DC_{idp} = \emptyset. \]
We use \texttt{isPrimitiveWF}(W) to denote the above predicate.

Intuitively, primitive workflow is a black box that has inputs and outputs and that represents an atomic component, such as Web service. Primitive workflows are used by other workflows as building blocks. Workflows such as $WS_1$, $WS_2$, \texttt{Not}, \texttt{Increment}, \texttt{Decrement}, \texttt{Sqrt}, \texttt{Square}, \texttt{Mean}, and \texttt{Divide} in Fig. 1 and Fig. 2 are primitive workflows.

**Definition 2.5 (Composite workflow).** A workflow $W$ is composite if and only if it contains at least one reusable component (i.e. $W.W^* \neq \emptyset$) connected to ports and/or data products. Formally, $W$ is composite iff
\[ (W.W^* \neq \emptyset \land W.IP \neq \emptyset \land W.OP \neq \emptyset \land W.DC_{in} \neq \emptyset \land W.DC_{out} \neq \emptyset) \lor (W.W^* \neq \emptyset \land W.OP \neq \emptyset \land W.DP \neq \emptyset \land W.DC_{idp} \neq \emptyset \land W.DC_{out} \neq \emptyset) \]
We use \texttt{isComposite}(W) to denote the above predicate. All workflows $W_s, W_a, W_b, \ldots, W_g$ in Fig. 1 and Fig. 2 are composite.

Intuitively, reusable workflows are primitive or composite tasks that can be reused as building blocks of more complex workflows. They are not executable as at least some of their input ports are not bound. Workflows $W_b, W_e$ and $W_g$ in Fig 2 are reusable. Workflow $W_b$ is reused inside $W_c$. Executable workflows, on the other hand have all input data needed to perform computation. Workflows $W_s, W_a, W_c, W_d, W_f, W_g$ in Fig. 1 and Fig. 2 are executable. Each executable workflow must contain at least one component and one data product connected to it. Thus, every executable workflow is composite. The opposite is not true, as composite workflow may be reusable (e.g., $W_b$), i.e. have input port(s) instead of concrete data product(s).

### 3 Workflow Expressions

We rely on simply typed lambda calculus [22] enriched with a set of primitive types as a formal framework to reason about the behavior of workflows. For example, expression \("\lambda x:\text{Int}. \text{Increment } x\)" is a function, or abstraction, that takes one integer argument, and returns its value increased by 1. $x$ is the abstraction name and \("\text{Increment } x\)" is the expression of this abstraction. The expression \("\text{Increment } 3\)" is an application, which evaluates to 4.

**Definition 3.1 (Workflow expression).** Given a workflow $W$, its expression $expr$ is a lambda expression that represents computation performed by $W$. If $W$ is reusable, $expr$ is an abstraction. If $W$ is executable, $expr$ is an application.

In this paper, we present our \texttt{translateWorkflow} function outlined in Algorithm 1, that given a workflow $W$, translates it into an equivalent lambda expression which performs the same computations and produces the same result as $W$. For simplicity we assume that workflow diagrams are drawn horizontally with data flowing from left to right (see Fig. 1, 2). Given a workflow $W$, our \texttt{translateWorkflow} algorithm translates all components in $W$ into lambda functions, and builds an expression whose structure corresponds to composition of components in $W$. Each connection between two components becomes an application in the equivalent lambda expression.
**Algorithm 1. Translating workflows into lambda expressions**

1: function translateWorkflow
2: input: workflow W
3: output: lambda expression for W
4: if isPrimitiveWF(W) /* If W is primitive, return its id */ then return W.id
5: else /* Otherwise, W is composite (reusable or executable), translate it recursively into lambda expression: */
6: /* First, find component in W.W* that performs the very last computational step (componentProducingFinalRes): */
7: let outputPortsOfDCmid be an empty set
8: for each ((wj, opj), (wk, ipj)) ∈ W.DCmid do
9: add opj to outputPortsOfDCmid
10: end for
11: for each W' ∈ W.W* do
12: if W'.OP ⊑ OutputPortsOfDCmid
13: then componentProducingFinalRes = W'
14: end if
15: end for
16: /* Build the list of expressions that serve as arguments for componentProducingFinalRes: */
17: listOfArguments = ‘’
18: for each (id, type) ∈ componentProducingFinalRes.IP do
19: listOfArguments += getInputExpression(W, componentProducingFinalRes, id) + ‘ ‘
20: end for
21: if W is reusable //|W.DCin | > 0
22: /* translate it into lambda abstraction: */
23: then
24: listOfNames = ‘’
25: for each (id, type) ∈ W.IP do
26: listOfNames += ‘λx’ + id + ‘:’ + type + ‘. ’
27: end for
28: return ‘(’ + listOfNames + translateWorkflow(componentProducingFinalRes)
29: ‘ ‘ + listOfArguments + ‘)’
30: else /* W is executable, thus translate it into a lambda application: */
31: return translateWorkflow(componentProducingFinalRes) + ‘ ‘ + listOfArguments;
32: end if
33: end if
34: end function
We accommodate composite workflows (containing sub-workflows) nested inside each other to arbitrary degree via recursive calls to `translateWorkflow` function that translates all sub-workflows at each level of nesting (depth-wise translation). We also accommodate arbitrary workflow compositions within the same level of nesting (flat compositions) by recursively calling `getInputExpression` function outlined in Algorithm 2, that iterates over and translates all the connected components by backtracking along the data channels from right to left (breadth-wise translation). Thus, our two algorithms together cover the full range of possible workflow compositions. We now provide a

**Algorithm 2.** Algorithm for obtaining lambda expressions representing inputs at certain workflow ports

```
1: function getInputExpression
2:   input: workflow W, constituent component c, input port ip_m
3:   output: lambda expression that serves as input argument of port W.id.
4:   /* first, if there is a data product dp_i in W.DP connected to port ip_m, return dp_i.id of that
data product: */
5:   for each (dp_i, (w_j, ip_k)) ∈ W.DC_out do
6:     if w_j.id = c.id and ip_k.id = ip_m.id then return dp_i.id end if
7:   end for
8:   /* if there is an input port ip_j in W.IP connected to port ip_m, return variable named “x” +
ip_j.id */
9:   for each (ip_j, (w_k, ip_l)) ∈ W.DC_in do
10:     if w_k.id = c.id and ip_l.id = ip_m.id then return “x” + ip_j.id end if
11:   end for
12:   /* if there is another constituent workflow w_i whose output is connected to ip_m, construct
the list of input arguments (expressions) of w_i and return application of these arguments to
w_i: */
13:   listOfArguments = “”
14:   for each ((w_i, op_j), (w_k, ip_l)) ∈ W.DC_mid do
15:     if w_k.id = c.id and ip_l.id = ip_m.id then
16:       for ip_q ∈ w_i.IP do
17:         listOfArguments += getInputExpression(W, w_i, ip_q) + “ ”
18:       end for
19:     return “(” + translateWorkflow(w_i) + “ ” + listOfArguments + “)”
20:   end if
21:   end for
22: return “error - cannot obtain input expression”
23: end function
```

We accommodate composite workflows (containing sub-workflows) nested inside each other to arbitrary degree via recursive calls to `translateWorkflow` function that translates all sub-workflows at each level of nesting (depth-wise translation). We also accommodate arbitrary workflow compositions within the same level of nesting (flat compositions) by recursively calling `getInputExpression` function outlined in Algorithm 2, that iterates over and translates all the connected components by backtracking along the data channels from right to left (breadth-wise translation). Thus, our two algorithms together cover the full range of possible workflow compositions. We now provide a
together cover the full range of possible workflow compositions. We now provide a walk-through example of how a workflow $W_d$ is translated into an equivalent lambda expression.

**Example 3.2 (Translating workflow $W_d$ into an equivalent lambda expression).** Consider a workflow $W_d$ in Fig. 2. When the function $\text{translateWorkflow}(W_d)$ is called, it first checks whether $W_d$ is primitive, and because it is not, the else clause is executed (lines 5-34). $\text{translateWorkflow}$ first determines that the component producing final result of the entire workflow $W_d$ is $\text{Sqrt}$ and stores it in the $\text{componentProducingFinalRes}$ variable (line 14). Next, because $\text{Sqrt}$ has a single input, for loop in lines 19-21 executes once, calling the function $\text{getInputExpression}(W_d, \text{Sqrt}, \text{ip}_7)$, whose output “$(\text{Mean dp}_0 \text{ dp}_1 \text{ dp}_2)$” is stored into a string $\text{listOfArguments}$. Next, $\text{translateWorkflow}$ checks whether $W_d$ is reusable (line 22), and because it is not it returns the application of workflow expression for $\text{Sqrt}$ component to the list of arguments obtained earlier (line 32). Since $\text{Sqrt}$ is a primitive workflow, $\text{translateWorkflow}(\text{Sqrt})$ returns its name “$\text{Sqrt}$”. Thus, the final result of the translation is “$\text{Sqrt} (\text{Mean dp}_0 \text{ dp}_1 \text{ dp}_2)$”.

**Example 3.3 (lambda expressions for workflows $W_s, W_a, W_b, \ldots, W_g$).** We provide lambda expressions obtained by calling our $\text{translateWorkflow}$ algorithm on each workflow in Fig. 1, 2:

- $W_s : \text{WS}2 \ (\text{WS}_1 \text{ dp}_0)$
- $W_a : \text{Increment} \ (\text{Not } \text{dp}_0)$
- $W_b : \lambda x_0 : \text{Bool} \ . \ \text{Increment} \ (\text{Not } x_0)$
- $W_c : W_b \ \text{dp}_0 \ = \ (\lambda x_0 : \text{Bool} \ . \ \text{Increment} \ (\text{Not } x_0)) \ \text{dp}_0$
- $W_d : \text{Sqrt} \ (\text{Mean } \text{dp}_0 \ \text{dp}_1 \ \text{dp}_2)$
- $W_e : \lambda x_0 : \text{Int} \ . \ \lambda x_1 : \text{Int} \ . \ \lambda x_2 : \text{Int} \ . \ (\text{Sqrt} \ (\text{Mean } x_0 x_1 x_2))$
- $W_f : \text{Divide} \ (\text{Increment} \ (\text{Square } \text{dp}_0)) \ (\text{Decrement} \ (\text{Square } \text{dp}_0))$
- $W_g : \lambda x_0 : \text{Int} \ . \ \text{Divide} \ (\text{Increment} \ (\text{Square } x_0)) \ (\text{Decrement} \ (\text{Square } x_0))$

Note that executable workflows ($W_s, W_a, W_c, W_d, W_f$) are translated into lambda applications, whereas reusable ones ($W_b, W_e, W_g$) into lambda abstractions. Ports are translated into variables, e.g. port $\text{ip}_0$ appears as $x_0$ in the corresponding expression. We require that the workflow expression is flat, i.e. constituent components’ ids are replaced with their translations (see expression for $W_c$). Thus, a workflow expression only contains port variables, names of primitive workflows, and data products.
4 Type System for Scientific Workflows

For interoperability, we adopt the type system defined in the XML Schema language specification [23]. This allows us to mediate WSDL-based Web services since their input and output types are described in WSDL documents according to the XSD format. While our approach can accommodate all types defined in [23], in this paper we focus on the set of types that are most relevant to the scientific workflow domain.

\[
T ::= T_{PRIM} | T_{XSD} | T \rightarrow T \\
T_{PRIM} ::= \text{String} | \text{Decimal} | \text{Integer} | \text{NonPositiveInteger} | \text{NegativeInteger} | \text{NonNegativeInteger} | \text{UnsignedLong} | \text{UnsignedInt} | \text{UnsignedShort} | \text{UnsignedByte} | \text{Double} | \text{PositiveInteger} | \text{Float} | \text{Long} | \text{Int} | \text{Short} | \text{Byte} | \text{Bool} \\
T_{XSD} ::= \{ e : T_{PRIM} \} | \{ e : T_{XSD_i} \} \quad i = 1 \ldots n 
\]

In our approach we allow primitive types (\(T_{PRIM}\)), XSD types (\(T_{XSD}\)), and arrow types (\(T \rightarrow T\)). A primitive type, such as \(\text{Int}\) or \(\text{Boolean}\) describes an atomic value. An XSD Type consists of an element name \(e\) and either a primitive type or an ordered set of other XSD types as detailed in the following example.

**Example 4.1 (XSD Type).** Consider an XML document \(d_{phd}\) shown in Fig. 3 (top left). Here we denote its XSD type as \(T_{phd}\). It consists of a name \(\text{gradStudent}\) and an ordered set of three children, each of which is another XSD type – \(\{\text{major}:\text{String}\}\), \(\{\text{gpa}:\text{Float}\}\), and \(\{\text{dissertTitle}:\text{String}\}\), as specified in Fig. 3. The first child has a name \(\text{major}\) and a type \(\text{String}\).

---

1 Although the two documents in Fig. 3 do not come from the scientific workflow domain, we use them in the paper to improve readability.
In this work we adhere to such notation for describing XSD types due to its conciseness compared to traditional XML Schema syntax. To improve readability, when discussing nested XSD types we omit curly braces at some levels of nesting. For simplicity, we focus on XML elements and do not explicitly model attributes. Since in XML each attribute belongs to a parent element, it can be viewed as a special case of an element without children.

The type constructor $\rightarrow$ is right-associative, i.e. the expression $T_1 \rightarrow T_2 \rightarrow T_3$ is equivalent with $T_1 \rightarrow (T_2 \rightarrow T_3)$. This type constructor is useful in defining types of reusable workflows. For example, the workflow $W_b$ has type $\text{Bool} \rightarrow \text{Int}$, since it expects boolean value as input and produces integer value as output. Workflow $W_e$ has the type $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Double}$. The type of an executable workflow is simply the type of its output, e.g., type of $W_a$ is $\text{Int}$.

We now introduce the notion of subtyping which is based on the fact that some types describe larger sets of values than others. For example, while the type $\text{Int}$ describes whole numbers in the range $[-2,147,483,648, 2,147,483,647]$, the type $\text{Decimal}$ describes infinite set of whole numbers multiplied by non-positive power of ten [23]. Thus, the set of values associated with the type $\text{Int}$ is a subset of values associated with the type $\text{Decimal}$, or, in other words, the type $\text{Decimal}$ describes larger set of values than $\text{Int}$ does. Therefore, it is safe to pass an $\text{Int}$ argument to a workflow expecting a $\text{Decimal}$ value as input.

Intuitively, given two types $S$ and $T$, $S$ is a subtype of $T$ (denoted $S <: T$), if all values of type $S$ form a subset of values of type $T$. Consider Table 1 listing type definitions for the primitive types adopted from [23]. We use $\mathbb{Z}$ to denote a set of all integers, i.e., $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, a commonly accepted notation. While [23] includes positive and negative infinity in value spaces of Double and Float, we omit them in our type system since they are not encountered in practical workflow executions. We also leave out special cases such as not-a-number (NaN) values.

Based on the type definitions adopted from [23] summarized here in the Table 1 we define a set of inference rules specifying subtype relationships between primitive types. For example, it is easy to see from the Table 1 that the set of $\text{Byte}$ values is a subset of $\text{Short}$ values, yielding a rule $\text{Byte} <: \text{Short}$. Other subtyping rules are shown in Fig. 4.

Fig. 5 shows the subtyping DAG that visualizes the subtype relationships (edges) between primitive types (nodes). A type $S$ is a subtype of $T$ is there is a path from the node $S$ to the node $T$. 
Table 1. Scientific Workflow Type System adopted from [23].

<table>
<thead>
<tr>
<th>data type</th>
<th>definition</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>set of finite-length sequences of characters</td>
<td>N/A</td>
</tr>
<tr>
<td>Decimal</td>
<td>{ a</td>
<td>a = i \times 10^n, i \in \mathbb{Z}, n \in \mathbb{Z}, and n \geq 0 }</td>
</tr>
<tr>
<td>Integer</td>
<td>\mathbb{Z}</td>
<td>(-\infty; +\infty)</td>
</tr>
<tr>
<td>NonPositiveInteger</td>
<td>{ a</td>
<td>a \in \mathbb{Z}, and a \leq 0 }</td>
</tr>
<tr>
<td>NegativeInteger</td>
<td>{ a</td>
<td>a \in \mathbb{Z}, and a \leq -1 }</td>
</tr>
<tr>
<td>NonNegativeInteger</td>
<td>{ a</td>
<td>a \in \mathbb{Z}, and a \geq 0 }</td>
</tr>
<tr>
<td>PositiveInteger</td>
<td>{ a</td>
<td>a \in \mathbb{Z}, and a \geq 1 }</td>
</tr>
<tr>
<td>UnsignedLong</td>
<td>{ a</td>
<td>a \in \mathbb{Z}, and a \geq 0, and a \leq 18446744073709551615 }</td>
</tr>
<tr>
<td>UnsignedInt</td>
<td>{ a</td>
<td>a \in \mathbb{Z}, and a \geq 0, and a \leq 4294967295 }</td>
</tr>
<tr>
<td>UnsignedShort</td>
<td>{ a</td>
<td>a \in \mathbb{Z}, and a \geq 0, and a \leq 65535 }</td>
</tr>
<tr>
<td>UnsignedByte</td>
<td>{ a</td>
<td>a \in \mathbb{Z}, and a \geq 0, and a \leq 255 }</td>
</tr>
<tr>
<td>Double</td>
<td>{ a</td>
<td>a = m \times 2^e, m \in \mathbb{Z}, and</td>
</tr>
<tr>
<td>Float</td>
<td>{ a</td>
<td>a = m \times 2^e, m \in \mathbb{Z}, and</td>
</tr>
<tr>
<td>Long</td>
<td>{ a</td>
<td>a \in \mathbb{Z}, and a \geq -9223372036854775808, and a \leq 9223372036854775807 }</td>
</tr>
<tr>
<td>Int</td>
<td>{ a</td>
<td>a \in \mathbb{Z}, and a \geq -2147483648, and a \leq 2147483647 }</td>
</tr>
<tr>
<td>Short</td>
<td>{ a</td>
<td>a \in \mathbb{Z}, and a \geq -32768, and a \leq 32767 }</td>
</tr>
<tr>
<td>Byte</td>
<td>{ a</td>
<td>a \in \mathbb{Z}, and a \geq -128, and a \leq 127 }</td>
</tr>
<tr>
<td>Bool</td>
<td>{ true, false }</td>
<td>[0; 1]</td>
</tr>
</tbody>
</table>

Decimal <: String
Integer <: Decimal
NonPositiveInteger <: Integer
NegativeInteger <: NonPositiveInteger
Long <: Integer
Int <: Long
Short <: Int
Byte <: Short
Bool <: Byte
Double <: Decimal
Float <: Double
Long <: Float
UnsignedInt <: Long
UnsignedShort <: Int
UnsignedByte <: Short
Bool <: UnsignedByte
NonNegativeInteger <: Integer
UnsignedLong <: NonNegativeInteger
UnsignedInt <: UnsignedLong
UnsignedShort <: UnsignedInt
UnsignedByte <: UnsignedShort
UnsignedLong <: Float
PositiveInteger <: NonNegativeInteger

Fig. 4. Subtyping rules for primitive types.
Similar intuition about subtyping applies to the structured types, such as XSD types. All the documents of the type \{a: Int\} form a subset of documents associated with the type \{a: Decimal\}.

Consider the two XML documents shown in Fig. 3. The type \(T_{phd}\) describes a set of XML documents with the root element \(gradStudent\) that has at least three children named \(major\), \(gpa\) and \(dissertTitle\) of types \(String\), \(Float\) and \(String\) respectively. Type \(T_{grad}\) on the other hand is less demanding as it requires only two child elements (\(major\) and \(gpa\)). Because \(T_{phd}\) is more specific, documents described by it form a subset of documents described by \(T_{grad}\), as shown in Fig. 3. Thus, it is safe to pass an argument of type \(T_{phd}\) to a workflow expecting an input of type \(T_{grad}\) since it will contain all the data needed by this workflow plus some extra, which can be ignored.

More generally, an XSD type \(S\) is a subtype of another XSD type \(T\) if \(S\)’s children form a superset of \(T\)’s children. Besides, if for each pair of corresponding children of \(S\) and \(T\) \(c_s\) and \(c_t\), \(c_s \prec c_t\) is true, then the statement \(S \prec T\) still holds. For example, if \(T_{grad.gpa}\) was of type \(Decimal\), \(T_{phd} \prec T_{grad}\) would still be true since \(Float \prec Decimal\).

Such view of subtyping, based on the subset semantics, is called the principle of safe substitution. Workflows \(W_o\), \(W_a\), \(W_b\), and \(W_e\) in Fig. 1, 2 are composed by this principle.

We formalize the subtype relation as a set of inference rules used to derive statements of the form \(S \prec T\), pronounced “\(S\) is a subtype of \(T\)”, or “\(T\) is a supertype of \(S\)”, or “\(T\) subsumes \(S\)”, where \(S\) and \(T\) are two types. As shown in Fig. 6, the first two rules (S-Refl, and S-Trans) state that the subtype relation is reflexive and transitive. They are then followed by a set of rules for primitive data types (collectively labeled S-Prim) discussed earlier.
We also include a rule $S$-XSD that formalizes the intuitive notion of subtyping discussed above for XSD types. This rule can be used, for example, to infer that the type $T_{\text{phd}}$ in Fig. 3 is a subtype of $T_{\text{grad}}$.

**Definition 4.1 (Subtype relation).** A subtype relation is a binary relation between types, $S <: T$ that satisfies all instances of the inference rules in Fig. 6.

Thus, according to the above definition the existence of the subtyping derivation concluding that $S <: T$ shows that $S$ and $T$ belong to the subtype relation. We now demonstrate how subtyping inference rules are used to derive statements of the form $S <: T$.

**Example 4.2 (Subtyping derivation inferring $T_{\text{phd}} <: T_{\text{grad}}$).** Fig. 7 (a) shows subtyping derivations concluding that the two types $T_{\text{phd}}$ and $T_{\text{grad}}$ in Fig. 3 belong to the subtype relation, i.e. $T_{\text{phd}} <: T_{\text{grad}}$. Each derivation step is labeled with the corresponding subtyping inference rule. In Fig. 7(a) we first note that the set $\{\text{major: String, gpa: Float}\}$ is a subset of $\{\text{major: String, gpa: Float, dissertTitle: String}\}$. We then show that $\{\text{major: String}\}$ is a subtype of $\{\text{major: String}\}$ using S-Refl rule. Similarly we show that $\{\text{gpa: Float}\}$ is a subtype of $\{\text{gpa: Float}\}$. These three statements together form a premise from which we can infer that $\{\text{gradStudent: major: String, gpa: Float, dissertTitle: String}\} <: \{\text{gradStudent: major: String, gpa: Float}\}$ based on the rule S-XSD as shown in Fig. 7 (a). This derivation formalizes the intuition that if a workflow can handle XML documents describing graduate students it can certainly handle documents describing PhD students.
Example 4.3 (Subtyping derivation inferring $T_1 <: T_2$). Fig. 7 (b) shows a subtyping derivation inferring that the two types $T_1$ and $T_2$ in Fig. 1 belong to the subtype relation, i.e. $T_1 <: T_2$. As shown in the figure, here we use four statements to form a premise from which we derive that $T_1 <: T_2$ according to the rule S-XSD.

In practice, the need arises to algorithmically determine whether for the two given types $S$ and $T$ the statement $S <: T$ is true. To this end, we now present a function that given two types $S$ and $T$ returns true if $S <: T$ and false otherwise. The function subtype is outlined in Algorithm 3. An XSD type $T$ is a data structure containing element name $e$ and an ordered set of children $T.children$. If $|T.children| > 1$, then each element in $T.children$ is another XSD type. If $|T.children| = 1$, then a single child ($T.children[0]$) is either a primitive type or an XSD type. We assume the existence of several functions that are described as follows. The function isPrimitive returns true if $T$ is a primitive type and false otherwise. The function isXSDType checks whether a given type is an XSD type. The function findChildWithName returns an item $c$ from the set of XSD types $E$ such that $c.e = name$. Finally, the function subtypePrim embodies rules S-Refl, S-Trans, and S-Prim by returning true if two given primitive types belong to the subtype relationship. For example, subtypePrim(Float, Int) returns true, whereas subtypePrim(Float, Double) returns false. As all four of these functions are trivial we omit their details for brevity.

Example 4.4 (Determining that $T_{phd} <: T_{grad}$ using the subtype function). When the function subtype($T_{phd}$, $T_{grad}$) is invoked, it first checks whether the two types are equal (line 4), and since $T_{phd} \neq T_{grad}$ it proceeds to line 5 to check whether both types are primitive. Since both $T_{phd}$ and $T_{grad}$ are XSD types (i.e. not primitive) the algorithm enters the else if clause (lines 7-28). It first ensures that both element names are the same (gradStudent) (line 8). It then checks whether $T_{phd}$ and $T_{grad}$ are both simple types, i.e. they do not contain nested XSD types inside (lines 9-11). Since both $T_{phd}$ and $T_{grad}$ are complex types, the algorithm builds two sets of element names of children of both types (lines 16-22):

(childrenNamesOfS = {major, gpa, dissertTopic})
(childrenNamesOfT = {major, gpa})
Algorithm 3. Algorithm for checking whether two given types belong to the subtype relation.

1: function subtype
2: input: two types $S$ and $T$.
3: output: true if $S <: T$, otherwise false
4: // if the two types are equal, return true (S-Refl):
5: if $S = T$ then return true end if
6: //if $S$ and $T$ are primitive, call subtyping on prim. types (S-Prim):
7: if isPrimitive($S$) and isPrimitive($T$) then return subtypePrim($S$, $T$) end if
8: //check whether the rule S-XSD applies to $S$ and $T$.
9: //First, both element names must be the same for $S <: T$ to be true
10: if $S.e \neq T.e$ then return false end if
11: if isXSDType($S$) and isXSDType ($T$) then
12: let childrenNamesOfS, childrenNamesOfT be two empty sets
13: for each child $\in S.children$ add child.e to childrenNamesOfS
14: end for
15: for each child $\in T.children$ add child.e to childrenNamesOfT
16: end for
17: //children names in $S$ must be a superset of those in $T$ and each child in $S$ must be a
18: //subtype of the //corresponding child in $T$:
19: if childrenNamesOfT $\subseteq$ childrenNamesOfS then
20: for each childOfT $\in T.children$
21: let childOfS = findChildWithTheName(childOfT.e, S.children)
22: if $\neg$ subtype(childOfS, childOfT) then return false end if
23: end if
24: end for
25: end if
26: return true
27: end if
28: end if
29: return false

It then checks whether the set childrenNamesOfT is a subset of childrenNamesOfS (line 19) and because it is, the algorithm iterates over every child in $T.children$, finds corresponding child from $S.children$ (i.e. child with the same element name) and checks whether they belong to the subtype relation (lines 20-25). If at least one pair of corresponding children did not satisfy the subtype relation, algorithm would return false. For example, if $T_{phd.gpa}$ was Decimal, the algorithm would detect it and return false.
since \{gpa:Decimal\} is not subtype of \{gpa:Float\} (lines 22-24). However, since every pair of respective children satisfies subtype relation, after iterating over each pair the algorithm returns \textit{true} (line 26). Note that the algorithm would still return \textit{true} if for example \(T_{\text{grad.gpa}}\) was of type \textit{Decimal} since \{gpa:Float\} \(<\) \{gpa:Decimal\}.

5 Typechecking Scientific Workflows

To determine whether a given workflow can execute successfully, we need to check whether connections between its components are consistent, i.e. each component receives input data in the format it expects. The expected format is constrained by a type declared in component’s specification. We formalize such consistency of connections through the notion of workflow \textit{well-typedness}. We check whether a workflow is well-typed by attempting to find its type.

Intuitively, we can derive the type of a workflow expression if we know the types of primitive workflows and data products involved in it. For example, it is easy to see that the expression \((\text{Increment } dp_0)\) has the type \textit{Int}, assuming \textit{Increment} expects integer argument and returns integer (formally, \textit{Increment:Int}→\textit{Int}) and \(dp_0\) is of type \textit{Int}. In other words, we can derive workflow type given a set of assumptions.

Typing derivation is done according to a set of inference rules (Fig. 8) for variables (T-Var), abstractions (T-Abs), and applications (T-App), as well as the rule for \textit{application with substitution} (T-AppS) that provides a bridge between typing and subtyping rules. Our inference rules for typing and subtyping are based on those from the classical theory of type systems [22], although modified to suit the scientific workflow domain and to ensure determinism of the typechecking algorithm presented later in this section.

In our rules, variable \(x\) represents a \textit{primitive object}, such as primitive workflow, port or data product, \(t\), \(t_{\text{arg}}\) and \(t_f\) are lambda expressions, and \(T\), \(T_1\), \(T_2\), \(T_{\text{in}}\) and \(T_{\text{out}}\) denote types. Set \(\Gamma = \{x_0:T_{p0}, x_1:T_{p1}, \ldots, x_n:T_{pn}\}\) is a \textit{typing context}, i.e. a set of assumptions about primitive objects and their types. The first rule (T-Var) states that variable \(x\) has the type assumed about it in \(\Gamma\). The second rule (T-Abs) is used to derive types of expressions representing reusable workflows. It states that if the type of expression with \(x\) plugged in is \(T_2\), then the type of abstraction, with the name \(x\) and expression \(t\) is \(T_1\to T_2\). The third rule (T-App) is used to derive types of applications, which represent data channels in workflows. The next rule (T-AppS) is necessary to typecheck workflows

\[
\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}
\]

\[
\frac{\Gamma \cup \{x : T_1\} \vdash t : T_2}{\Gamma \vdash \lambda x : T_1, t : T_1 \to T_2} \quad \text{(T-ABS)}
\]

\[
\frac{\Gamma \vdash t_{\ell} : T_{\text{in}} \to T_{\text{out}} \quad \Gamma \vdash t_{\text{arg}} : T_{\text{in}}}{\Gamma \vdash t_{\ell} t_{\text{arg}} : T_{\text{out}}} \quad \text{(T-APP)}
\]

\[
\frac{\Gamma \vdash t_{\ell} : T_{\text{in}} \to T_{\text{out}} \quad \Gamma \vdash t_{\text{arg}} : T_{\text{in}}}{\Gamma \vdash t_{\ell} t_{\text{arg}} : T_{\text{out}}} \quad \text{(T-APPS)}
\]

Fig. 8. Workflow typing rules.
with subtyping connections (shown dashed in Fig. 1, 2). We call such compositions workflows with subtyping. We show a concrete type derivation that uses the above rules in Example 5.3.

**Definition 5.1 (Workflow context).** Given a workflow \( W \), a workflow context \( Z \) is a set of all data products and primitive workflows used inside \( W \) (at all levels of nesting) and their respective types.

**Definition 5.2 (Well-typed workflow).** A workflow \( W \) is well-typed, or typable, if and only if for some \( T \), there exists a typing derivation that satisfies all the inference rules in Fig. 8, and whose conclusion is \( Z \vdash W : T \), where \( Z \) is a workflow context for \( W \).

**Example 5.3 (Typing derivation for workflow \( W_a \)).** Consider the workflow \( W_a \) shown in Fig. 2. Its workflow expression is \( \text{Increment} \ (\text{Not} \ dp_0) \). \( W_a \)'s workflow context \( Z \) is a set \{\( \text{Increment} : \text{Int} \rightarrow \text{Int}, \ \text{Not} : \text{Bool} \rightarrow \text{Bool}, \ dp_0 : \text{Bool} \\}. A typing derivation tree for this workflow is shown in Fig. 9.

![Fig. 9. Typing derivation for workflow \( W_a \).](image)

Similarly to the subtyping derivations in Fig. 7, each step here is labeled with the corresponding typing inference rule. Derivation holds for \( \Gamma = Z \). According to Definition 5.2, existence of typing derivation with the conclusion \( \{\text{Increment} : \text{Int} \rightarrow \text{Int}, \ \text{Not} : \text{Bool} \rightarrow \text{Bool}, \ dp_0 : \text{Bool} \}\vdash \text{Increment} (\text{Not} \ dp_0) : \text{Int} \), proves that \( W \) is well-typed.

**Example 5.4.** (Typing derivation for workflow \( W_s \)). Consider a workflow \( W_s \) in Fig. 1 whose workflow expression is \( WS_2 (WS_1 dp_0) \). Its workflow context \( Z \) is a set \{\( WS_1 : \{\text{String} \rightarrow T_1\}, WS_2 : \{T_2 \rightarrow \text{Int}\}, dp_0 : \text{String} \}\}, where

\[
T_1 = \{\text{data}: \{\text{experimId}: \text{String}, \ \text{concentr}: \text{Float}, \ \text{degree}: \text{Int}, \ \text{model}: \{\text{response}: \text{Double}, \ \text{hillSlope}: \text{Double}\}\}\},
\]

and

\[
T_2 = \{\text{data}: \{\text{degree}: \text{Int}, \ \text{model}: \{\text{response}: \text{Double}, \ \text{concentr}: \text{Double}\}\}\}
\]

The typing derivation for \( W_s \) is shown in Fig. 10. We use \( C :: T_1 <: T_2 \) as a shorthand to denote a subtyping derivation with the conclusion \( T_1 <: T_2 \). The complete subtyping derivation is shown in Fig. 7 (b). Because we can derive the type of \( W_s \) using the typing inference rules, this workflow is well-typed, according to the Definition 5.2.

![Fig. 10. Typing derivation for workflow \( W_s \).](image)

We now introduce the generation lemma that we use to design our typechecking function. Generation lemma captures three observations about how to typecheck a given workflow. Each entry is read as “if workflow expression has the type \( T \), then its subexpressions must have types of these forms”. Each observation inverses the corresponding rule in Fig. 8 by stating it “from bottom to top”. Note that for T-Abs we add to the context variable-type pair for name \( x \), which is given explicitly in the
abstraction.

**Lemma 5.5 (Generation lemma).**

GL1. \( \Gamma \vdash x:T \Rightarrow x:T \in \Gamma \) /* inverses T-Var */

GL2. \( \Gamma \vdash (\lambda x:T_1. t) : T \Rightarrow \exists T_2 (T = T_1 \rightarrow T_2 \wedge (\Gamma \cup \{x:T_1\} \vdash t:T_2)) \) /* inverses T-Abs */

GL3. \( \Gamma \vdash t\_in \Rightarrow \exists T\_out (\Gamma \vdash t\_in \rightarrow T\_out \wedge ((\Gamma \vdash t\_arg:T\_in) \vee \exists T_1 (\Gamma \vdash t\_arg:T_1 \wedge T_1 <: T\_in))) \) /* inverses T-App and T-AppS */

**Proof:** GL1 - by contradiction. Assume \( \Gamma \vdash x:T \), and \( x:T \notin \Gamma \). Since \( \Gamma \vdash x : T \), there must be a typing derivation satisfying inference rules in Fig. 8 with the conclusion \( \Gamma \vdash x : T \). Rules T-Abs and T-App and T-AppS cannot be used to derive the type of \( x \), since neither of them deduces a type of a primitive object. The rule T-Var is also not applicable since \( x:T \in \Gamma \) is false. Thus, there exists no derivation with the conclusion \( \Gamma \vdash x : T \), and hence \( \Gamma \vdash x : T \) cannot be true, which is a contradiction. GL2 and GL3 can be proved similarly by contradiction. \( \square \)

In practice, to reason about workflow behavior we need a deterministic algorithm to derive the type of \( W \). To this end, we now present the `typecheckWorkflow` function outlined in Algorithm 4. Given a workflow \( W \), it derives \( W \)'s type from the primitive objects inside \( W \) according to the typing rules in Fig. 8. This function is a transcription of the generation lemma (Lemma 5.5) that performs backward reasoning on the inference rules. Each recursive call of `typecheckWorkflow` is made according to the corresponding entry (GLx) of the generation lemma. We assume that methods \( \Gamma.getBinding(name) \) and \( \Gamma.addBinding(name, type) \) get the type of a given variable and add the variable-type pair to the context \( \Gamma \) respectively, \( abstraction.name, abstraction.nameType \) and \( abstraction.expression \) return name, type of name variable and expression of the given abstraction respectively. \( application.a \) and \( application.f \) return function and argument of an application respectively.

### 6 Automatic Coercion in Workflows

While workflow welltypedness is a necessary condition for the successful execution, it is by no means sufficient. As we explain in the following, in order to run properly, workflows with subtyping need to have shims at every subtyping connection that explicitly convert data products to their target types.

Consider the workflow \( W_b \) shown in Fig. 2. Although the `Bool` type is a subtype of `Int`, data products of these two types may have entirely different physical representations in workflow management systems. In particular, the workflow engine may use two different classes BoolDP and IntDP to represent data products holding values of types `Bool` and `Int`. If neither of the two classes is a subclass of the other, casting BoolDP to IntDP is impossible and hence using BoolDP in place of IntDP will result in runtime error during workflow execution. To avoid such error, data products of type `Bool` need to be explicitly converted or coerced to those of type `Int`.

Similar reasoning applies to XML data products. As shown in Fig. 1, the dashed connection in workflow \( W_s \) links two ports whose types satisfy the subtype relationship \( (T_1 <: T_2) \). Nonetheless, sending \( d_1 \) as input for WS2 will cause an error if \( d_1 \) is not transformed appropriately to validate against the input schema of WS2.
Thus, to ensure successful evaluation, we adopt the so-called coercion semantics for workflows, in which we replace subtyping with runtime coercions that change physical representation of data products to their target types.

We express the coercion semantics for workflows as a function translateT that translates workflow expressions with subtyping into those without subtyping. In this paper, we use $C :: S <: T$ to denote subtyping derivation tree whose conclusion is $S <: T$ (as we did in the Example 5.4). Similarly, we use $D :: \Gamma \vdash t : T$ to denote typing derivation whose conclusion is $\Gamma \vdash t : T$. Given a subtyping derivation $C :: S <: T$, function translateS(C) returns a coercion (lambda expression) that converts data products of type $S$ into data products of type $T$.

---

**Algorithm 4. Typechecking of scientific workflows**

1. function typecheckWorkflow
2. input: workflow expression expr, context $\Gamma$
3. output: type of $W$
4. if expr is primitive object /*GL1: expr is a variable representing port, primitive workflow or data product*/ then
5. return $\Gamma$.getBinding(expr)
6. else if expr is abstraction /*GL2 */ then
7. let $\Gamma' = \Gamma$
8. $\Gamma'$.addBinding(expr.name, expr.nameType)
9. typeOfExpr = typecheckWorkflow (expr.expression, $\Gamma'$)
10. return expr.nameType $\to$ typeOfExpr
11. else if expr is application /*GL3 */ then
12. typeOfF = typecheckWorkflow(expr.f, $\Gamma$)
13. typeOfN = typecheckWorkflow(expr.n, $\Gamma$)
14. if typeOfF is of the form $T_0 \to T_1 \to \ldots \to T_n$, where $n > 0$ then
15. if subtype(typeOfN, $T_0$) then
16. return $T_1 \to \ldots \to T_n$
17. else
18. return "error: parameter type mismatch"
19. end if
20. else
21. return "arrow type expected"
22. end if
23. end if
24. end function
We denote function `translateS(C)` as `[[C]]` and define it in a case-by-case form:

\[
\begin{align*}
\text{T} & \vdash \text{T} \quad \text{S-REPL} \\
\text{Bool} & \vdash \text{Byte} \quad \text{S-Phy} \\
\text{Int} & \vdash \text{Long} \quad \text{S-Phy} \\
S & \vdash T \quad \text{S-Phy} \\
[C_1 : S : U & C_2 : U : T \triangleright \text{isPrimitive}(S) \quad \text{S-TRANS}] \\
C & \vdash T \quad \text{S-Phy} \\
\{e:S\} & \vdash \{e:T\} \quad \text{S-XSD} \\
\{S_{i\in 1..n}\} & \subseteq \{U_{j\in 1..n+k}\}, \quad n \geq 1, \quad k \geq 0, \\
\text{for each } i \in 1..n \quad C_i & \vdash T_i \quad \text{isPrimitive}(S_i) \\
\{e:U_{j\in 1..n+k}\} & \vdash \{e:T_{i\in 1..n}\} \quad \text{S-XSD}
\end{align*}
\]

where functions `wrap`, `getContent`, `compose`, and `extract` are defined below.

The function `wrap(e x)` encloses its input `x` in an XML element with the name `e`, e.g.,

\[
\text{wrap} \text{("concentr") 15.1} = <\text{concentr} 15.1</\text{concentr}>
\]

The function `getContent(x)` returns a simple content of an XML element `x`, e.g.,

\[
\text{getContent} <\text{concentr}>15.1<\text{concentr}> ) = 15.1
\]

The function `extract(e x)` extracts a child element of `x` named `e`, e.g.,

\[
\text{extract("response")}
\]

\[
\begin{align*}
\text{<model>}
& \quad \text{<response>40.5</response>} \\
& \quad \text{<hillslope>3.8</hillslope>}
\end{align*}
\]

\[
\text{</model>}
\]

\[
\text{=} \text{<response>40.5</response>}
\]

The function `compose(e x_1 x_2 ... x_i)` composes an XML element with the name `e` and children `x_1 x_2 ... x_i`, e.g.,

\[
\text{compose("data" <degree>25</degree>}
\]

\[
\begin{align*}
\text{<model>}
& \quad \text{<response>40.5</response> </model>}
& \quad <\text{concentr}15.1</\text{concentr}> = \\
& \quad <\text{data}>
& \quad \text{<degree>25</degree>}
& \quad \text{<model>}
& \quad \text{<response>40.5</response> }
& \quad \text{</model>}
& \quad <\text{concentr}15.1</\text{concentr}>
\end{align*}
\]

\[
\text{</data>}
\]
The first four cases describe how to translate subtyping derivations consisting of only one inference step, made using the rule S-Prim. The fifth case applies when S-Trans rule is used at the final step to infer subtype relationship between primitive types. The sixth case applies for derivation trees that use S-XSD rule and when isPrimitive(S) is true. For example it applies for the derivation concluding \{concentr:Float\} <: \{concentr:Double\}. Seventh case applies for all derivations whose last step is made using S-XSD rule and when isPrimitive(S) is false. It applies, for example to the derivation of \(T_{phd} <: T_{grad}\) shown in Fig. 7 (a).

Given a typing derivation \(D :: \Gamma : t : T\), function translateT(D) produces an expression similar to \(t\) but in which subtyping is replaced with coercions. We also denote translateT(D) as \([[D]]\). From the context, it will be clear which of the two functions is being used. Similarly, we define translateT by cases:

\[
\begin{align*}
&\Gamma \vdash \text{tVar} & = \text{tVar} \\
&\Gamma \vdash \text{tApp} & = \lambda \text{tApp} \\
&\Gamma \vdash \text{tArg} & = [[\Gamma]] [[D]] \\
&\Gamma \vdash \text{tArgS} & = [[\Gamma]] [[D]] \\
&\Gamma \vdash \text{tArgS} & = [[\Gamma]] [[D]] \\
\end{align*}
\]

Note that in the case of T-AppS rule, translateT calls translateS(C) to retrieve appropriate coercion and insert it into the application where subsumption took place. Thus, while translateT is used for typing derivations (e.g., Fig. 9), translateS is used for subtyping derivations (e.g., Fig. 7 a, b).

Example 6.1 (Inserting a primitive coercion into the workflow \(W_a\) using the function translateT). Consider the workflow expression \(\text{Increment (Not dp0)}\) which corresponds to the workflow \(W_a\) shown in Fig. 2. To inject coercions into it, we call function translateT. The function takes the typing derivation tree shown in Fig. 9 as input and produces a workflow expression with coercion inserted as output. The function evaluates as follows
The translation begins from the last derivation step in Fig. 9 and progresses from bottom to top. Because the rule T-AppS was used at the final inference step, the last case applies from the definition of translateT yielding an application \([D_{f1} :: \Gamma \vdash \text{Increment}: \text{Int} \to \text{Int}])\ (\[C :: \text{Bool} <: \text{Int} \] \[D_{arg1} :: \Gamma \vdash (\text{Not } dp_0) : \text{Bool}\] ) in which \(D_{f1} :: \Gamma \vdash \text{Increment}: \text{Int} \to \text{Int}\) and \(D_{arg1} :: \Gamma \vdash (\text{Not } dp_0) : \text{Bool}\) are replaced with the corresponding typing derivations for \(\text{Increment}\) and \((\text{Not } dp_0)\) respectively, and \(C :: \text{Bool} <: \text{Int}\) is replaced with subtyping derivation for \(\text{Bool} <: \text{Int}\). The function then calls itself recursively on \(D_{f1}\) and \(D_{arg1}\) and also calls translateS on \(C\). Since T-Var was used for the last inference step in \(D_{f1}\), translateT\((D_{f1})\) returns \(\text{Increment}\). In \(D_{arg1}\) on the other hand, T-App was used to make the last inference step and so the third case in translateT’s definition applies. Thus, translateT calls itself recursively on \(D_{f2} :: \Gamma \vdash \text{Not} : \text{Bool} \to \text{Bool}\) and on \(D_{arg2} :: \Gamma \vdash dp_0 : \text{Bool}\). In both calls the first case of translateT applies yielding \(\text{Not}\) and \(dp_0\) respectively. The call translateS\((C::\text{Bool} <: \text{Int})\) returns a coercion \(\text{Bool2Int}\) which corresponds to the fifth case in translateS’ definition since isPrimitive(Bool) is true. Thus the function translateT replaced subtyping in the typing derivation (i.e. \(\text{Bool} <: \text{Int}\)) with the coercion \(\text{Bool2Int}\) that converts \(\text{Bool}\) data products to \(\text{Int}\) data products. Coercion \(\text{Bool2Int}\) implemented as a primitive workflow is inserted dynamically at runtime and is transparent to the user.

**Example 6.2 (Inserting a composite coercion into the workflow \(W_s\) using the function translateT).** We now demonstrate how function translateT inserts coercion in the workflow expression \(WS_2\left(WS_1; dp_0\right)\) which corresponds to the workflow \(W_s\) shown in Fig. 1. translateT takes a typing derivation tree in Fig. 10 as input. The evaluation proceeds as follows

\[
\begin{align*}
\left[ D_{f1} :: \Gamma \vdash WS_2; T_2 \to \text{Int} \right. \\
\left. D_{arg1} :: \Gamma \vdash (WS_1; dp_0); T_1 \right. \\
\left. C :: T_1 <: T_2 \right] \\
\Gamma \vdash (WS_2 (WS_1; dp_0)) : \text{Int} \\
\left[ \text{T-AppS} \right] \\
= \left[ WS_2; T_2 \to \text{Int} \in \Gamma \right. \\
\left. \Gamma \vdash WS_2; T_2 \to \text{Int} \right. \\
\left. \text{T-Var} \right] \\
([C_{T_1,T_2}:T_1 <: T_2]) \\
\left[ D_{f2} :: \Gamma \vdash WS_1; \text{String} \to T_1 \right. \\
\left. D_{arg2} :: \Gamma \vdash dp_0; \text{String} \right. \\
\left. (WS_1; dp_0):T_1 \right] \\
= WS_2 \left( \text{compositeCoercion}_{T_1,T_2} \right. \\
\left. (WS_1; String) \to T_1 \in \Gamma \right. \\
\left. \Gamma \vdash WS_1; \text{String} \to T_1 \right. \\
\left. \text{T-Var} \right] \\
\left. \left( dp_0; \text{String} \in \Gamma \right. \right. \\
\left. \left. \Gamma \vdash dp_0; \text{String} \right. \right. \\
\left. \text{T-Var} \right] \\
= WS_2 \left( \text{compositeCoercion}_{T_1,T_2} \right. \\
\left. \left(WS_1; dp_0\right) \right)
\end{align*}
\]

, where \(\text{compositeCoercion}_{T_1,T_2}\) denotes the result of \([C :: T_1 <: T_2]\). The complete translation process yielding this result is shown in Fig. 11. Again, function translateT calls itself recursively at each step. Similarly to the previous example it also calls translateS on subtyping derivation tree inferring \(T_1 <: T_2\). This tree is shown in Fig. 7 (b) and is denoted here as \(C :: T_1 <: T_2\). First, because the S-XSD rule is used at the last inference step of \(C :: T_1 <: T_2\) and \(\neg \text{isPrimitive}([\text{degree} : \text{Int}])\) is true, the last case of translateS’s
Fig. 11. Translating subtyping derivation into a composite coercion using function \texttt{translateS}.

definition applies with

\[ S_i = \{ \text{degree: Int, model: \{response: Double, hillSlope: Double, concentr: Float\}} \}, \]
\[ U_j = \{ \text{experimId: String, concentr: Float, degree: Int, model: \{response: Double, hillSlope: Double\}} \} . \]

As shown in Fig. 11, the function \texttt{translateS} calls itself recursively on derivations \( C_1 \), \( C_2 \) and \( C_3 \). Because S-Refl rule was used in \( C_1 \), \( [C_1] \) yields the identity function \( \lambda x. x \), which
simply returns its argument. Thus application $\lambda x. x$ (extract degree $x$) evaluates to (extract degree $x$). The translation process eventually yields a lambda expression, which we denote as compositeCoercion$_{T1-T2}$ for convenience.

The role of compositeCoercion$_{T1-T2}$ is to transform XML documents produced as the output of $WS_1$ into documents that will validate against the input XSD schema of $WS_2$ which will allow $WS_2$ to execute properly. This enables safe execution of the workflow $W_a$. For example, when applied to the XML document $d_1$ in Fig. 1, this coercion extracts sub-elements of $d_1$, coerces them to the target types and composes the resulting elements into a new XML document of type $T_2$. The result is the document $d_2$. In particular, the coercion extracts degree element leaving it unchanged since its type is identical to that of the corresponding element in the target type $T_2$. It then extracts model and response elements and creates a new model element that only contains response element, leaving out the hillSlope, which is not part of $T_2$. The coercion also extracts element concentr, gets its simple content, converts it from Float to Double and wraps it back into $<concentr>$ tags. Finally, the coercion builds data element out of the three previously obtained elements - degree, model, and concentr. The resulting XML element validates against $WS_2$’s input schema, and hence $WS_2$ will now run without an error.

7 IMPLEMENTATION AND CASE STUDIES

We now present the new version of our VIEW system [25], in which we implement our automated shimming technique including the proposed workflow model, algorithms 1, 2, 3, and 4, simply typed lambda calculus, and our translation functions translateS and translateT.

Our new version of VIEW is web-based, with no installation required. Scientists access VIEW through a browser and compose scientific workflows from Web services, scripts, local applications, etc. A workflow structure is captured and stored in a specification document written in our XML-based language SWL. A workflow is executed by pressing the “Run” button in the browser. Once the “Run” button is pressed, our system inserts shims and executes the workflow. To avoid cluttering the workflow and help scientists focus on its functional components, inserted shims are hidden from the user.

7.1 Primitive Shimming in Workflow $W_a$

Fig. 12 displays the workflow $W_a$ from earlier examples, and a screenshot of the VIEW system dialog window showing $W_a$’s SWL (top left part of the dialog). Once the user has pressed the “Run” button the system uses Algorithm 1 (which calls Algorithm 2 as a subroutine), to translate the workflow into a typed lambda expression with subtyping (Step 1 in Fig. 12). It then typechecks $W_a$ using Algorithm 4. After VIEW ensures that $W_a$ is well-typed, using the function translateT (which in turn uses translateS) our system inserts coercions (workflows performing type conversion) into the workflow expression by translating it into lambda calculus without subtyping (Step
Fig. 12. Automatically inserting primitive shim in workflow $W_a$ using the VIEW system.

2 in Fig. 12). Particularly, subtyping $\text{Bool} <: \text{Int}$ is replaced with the corresponding coercion – $\text{Bool2Int}$. Finally, the obtained expression is translated into a runtime version of SWL (Step 3 in Fig. 12), which contains all the necessary shims. This runtime version of SWL is supplied to the workflow engine for execution.

Note that these three steps are fully automated and transparent to the user, who will see results of workflow execution upon pressing the “Run” button.

7.2 Composite Shimming in Workflow $W_s$

Workflow $W_s$ in Fig. 1 comes from the biological domain. Scientists use VIEW to gain insight into the behavior of the marine worm Nereis succinea [26]. Biologists study the effect of the pheromone excreted by female worms on the reproduction process. They compose a workflow that calculates the number of successful worm matings given a set of parameters, including pheromone concentration, initial degree of male worm, and a worm model. The model includes parameters describing worm’s behavior, such as maximum response to pheromone and steepness of the dose-response relationships (hill slope). Scientists use Web service $WS_1$ to retrieve a set of parameters and a worm model associated with a particular experiment. These data are fed into Web service $WS_2$ that simulates the movement and interaction between worms according to the supplied input parameters and model. The output of $WS_2$ is the number of successful worm matings, which is the final result of this workflow. However, to execute workflow $W_s$, the syntactic incompatibilities between WSDL interfaces of $WS_1$ and $WS_2$ must be resolved. We now demonstrate how our system accomplishes this by creating and inserting composite shim between $WS_1$ and $WS_2$. Fig. 13 illustrates workflow $W_s$ and a VIEW dialog window showing how shim is automatically inserted by our system. Similarly to the previous example, after translating $W_s$’s specification into the lambda expression (Step 1) VIEW replaces subtyping in this expression with runtime coercions (Step 2). Here the coercion is composite, i.e. a lambda expression consisting of multiple
functions. Finally, the obtained workflow expression that includes coercion is translated into the runtime version of the SWL specification (Step 3). The coercion becomes a composite shim, as shown in Fig. 13. During workflow execution, this shim decomposes a document that comes out of WS1 (i.e.  data>...<data>) into smaller pieces, reorders them to fit WS2’s input, converts them to the appropriate target types, and composes a new document out of the obtained elements. This new document validates against the input schema of WS2 allowing it to successfully compute the number of matings in a given experiment.

The inserted shim leaves out element “<hillSlope>3.8</hillSlope>”, which is not used by WS2. This reduces the size of the SOAP request sent to the server where WS2 is hosted by 9.3%. In other workflows, this portion may be much larger. Removing such unnecessary data from requests using our technique decreases the load on the network and on servers hosting Web services. Such efficient use of resources is especially important in workflows running in distributed environments.

The composite shim was generated solely based on the information in WSDL.
documents of $WS_1$ and $WS_2$. Our approach uses neither ontologies nor semantic annotations, nor does it require users to write transformation scripts.

8 RELATED WORK

Web Service composition plays a key role in the fields of services computing [1, 2, 3, 4, 5] and scientific workflows [6, 7, 8, 9]. The main challenge in the field of service composition is to mediate autonomous third-party Web services [10, 11, 12, 13]. Resolving interface incompatibilities between services by means of an intermediate component called shim is known as the shimming problem, widely recognized in the community [10, 11, 12, 13, 14, 15].

Some researchers have developed techniques to resolve Web services protocol mismatches [10, 27, 28]. These mismatches occur when the permitted sets of messages and/or their order differ in the protocols of Web services that are connected together. While such techniques focus on reconciling behavioral differences between Web services, (e.g., differences in number and/or order of messages) our work focuses on resolving the interface differences (e.g., different types of inputs/outputs).

Another category of mediation techniques relies on semantic annotations in Web services as well as domain models. For example, authors of [13, 16, 21] develop shims that transform XML documents whose elements are associated with semantic domain concepts, expressed in languages, such as OWL. Sellami et al. [20] address the shimming problem by using semantic annotations of Web services to find shims. Besides requiring composed Web services to be semantically annotated, this approach also expects Web service providers to supply all the necessary shims that are also annotated.

In contrast to [13, 16, 20, 21], our work focuses on the syntactic layer rather than the semantic layer, and relies solely on data types defined in WSDL schema. It applies regardless of whether semantic information was provided or not. Nonetheless, integrating our shimming technique would benefit the semantics-based solutions. Existing scientific workflow systems [29, 30, 31, 32] provide limited shimming capabilities i.e. shimming is either explicit or requires additional workflow configuration.

None of the above approaches (1) guarantees an automated solution with no human involvement, (2) makes shims invisible in the workflow specification, (3) provides a solution for arbitrary workflow (even within some well-defined class), (4) applies to both primitive and structured types. Our approach addresses all four issues.

To address these issues, in [12], we present a primitive workflow model and a workflow specification language that allows hiding shims inside task specifications. This paper improves our earlier work by proposing an approach that determines where a shim needs to be placed in the workflow, and inserts appropriate coercion in the workflow expression. Specifically, we choose typed lambda calculus [22] to represent workflows which is naturally suitable for dataflow modeling due to its functional characteristics [33]. While recognizing the importance of shims, [33] does not address the shimming problem. We formalize coercion in scientific workflows with
type-theoretic rigor [22, 34]. Existing typechecking techniques apply in contexts other than scientific workflows, e.g., Hindley-Milner algorithm [35] requires typed prefix to typecheck expressions with polymorphic types (not used in workflows) and therefore cannot be directly applied to typecheck workflow expressions. We present a concrete fully algorithmic solution and demonstrate its application to the specific workflow type system with primitive and structured (XSD) types.

To our best knowledge, this work is the first one to reduce the shimming problem to the coercion problem and to propose a fully automated solution. This paper extends [36] with the following additional contributions:

1. We add support for Web services mediation by extending our type system with XSD types defined in WSDL and by introducing a subtype algorithm to check the subtype relationship between types (Algorithm 3).
2. We define four new functions \texttt{wrap}, \texttt{getContent}, \texttt{compose}, and \texttt{extract} to generate composite coercions for XML data products.
3. We extend the definition of function \texttt{translateS} with two new cases to handle subtyping between Web service inputs/outputs.
4. We implement the proposed composite shimming technique for Web services in ourVIEW system and add a case study that demonstrates how our VIEW system generates and inserts a composite shim to mediate two Web services from the biological domain.

9 Conclusions and Future Work

In this paper, we first reduced the shimming problem to the runtime coercion problem in the theory of type systems. Second, we proposed a scientific workflow model and a notion of a well-typed workflow, and developed an algorithm to translate workflows into equivalent lambda expressions. Third, we developed an algorithm to typecheck scientific workflows. Fourth, we designed a function that inserts “invisible shims”, or runtime coercions that mediate Web services, thereby solving the shimming problem for any well-typed workflow. Finally, we implemented our automated shimming technique, including all the proposed algorithms, lambda calculus, type system, and translation functions in our VIEW system and presented two case studies to validate the proposed approach. In the future, we plan to develop more workflows to showcase our approach and use our shimming technique and to address the data variety challenge in Big Data.

References


Andrey Kashlev is a PhD candidate in computer science at Wayne State University. His research interests include Big Data, Scientific Workflows and Services Computing. He has published several papers in peer-reviewed international journals and conferences, including Data and Knowledge Engineering, International Journal of Computers and Their Applications and the International Conference on Services Computing. He is a member of IEEE.

Shiyong Lu is an associate chair and an associate professor in the Department of Computer Science, Wayne State University, and the director of the Big Data and Scientific Workflow Research Laboratory. Dr. Lu received his Ph.D. in computer science from the State University of New York at Stony Brook in 2002. Dr. Lu’s current research interests focus on Big Data and scientific workflows. Dr. Lu is an author of two books and over 80 articles published in various international journals and conferences, including IEEE Transactions on Services Computing (TSC), Data and Knowledge Engineering (DKE), IEEE Transactions on Knowledge and Data Engineering (TKDE). He is the founding chair of the IEEE International Workshop on Scientific Workflows (SWF) and a founding editorial board member of both International Journal on Semantic Web and Information Systems and International Journal of Healthcare Information Systems and Informatics. He is a senior member of the IEEE.

Artem Chebotko is an assistant professor in the Department of Computer Science at University of Texas - Pan American. He received his Ph.D. in Computer Science from Wayne State University. His research interests include semantic web data management, scientific workflow provenance metadata management, distributed data management and cloud computing, scientific workflows and services computing. He has published over 40 papers in refereed journals and conference proceedings and currently serves as an Editor of the International Journal of Cloud Computing and Services Science (IJ-CLOSER) and a Program Committee Member of several international conferences and workshops. He is a member of IEEE.