Coercion Approach to the Shimming Problem in Scientific Workflows

Andrey Kashlev, Shiyong Lu
Department of Computer Science
Wayne State University

Artem Chebotko
Department of Computer Science
University of Texas – Pan American

Abstract—When designing 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 three algorithms to typecheck workflows by first translating them into equivalent lambda expressions, 4) design two functions that together insert “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 functions in our VIEW system and present a case study to validate the proposed approach.

Keywords—shim; shimming problem; scientific workflows;

I. INTRODUCTION

Scientific workflows are becoming increasingly important to integrate, structure, and orchestrate a variety of heterogeneous services and applications into complex computational processes to enable and facilitate scientific discovery. Often times, composing autonomous third-party services and applications into workflows requires using intermediate components, called shims, to mediate syntactic and semantic incompatibilities between different heterogeneous components.

Consider, a workflow \( W_0 \) in Fig. 1 comprised of two web services – Not and Increment. Because the output of Not is a boolean value (true or false) while Increment is designed to process integer arguments, to execute the workflow we need to find and insert a shim that will resolve this incompatibility. Determining where the shim is needed, finding appropriate shim and inserting it is known as the shimming problem, whose significance has been widely recognized by the scientific workflow community [3-8]. 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 [10-11, 18]. 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 [3, 20] show that the percentage of shim components in workflows registered in myExperiment portal (www.myexperiment.org) has grown from 30% in 2009 [20] to 38% in 2012 [3]. 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., [9-12]) 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 [12]), 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, these efforts focus on shims for scientific data of a particular type, such as XML [10-12] or relational schemas [13], and cannot be generalized to handle all structured data types, let alone primitive types such as String or Double.

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 three algorithms to typecheck workflows by translating them into equivalent lambda expressions,
4. design two functions that together insert “invisible shims” (coercions) into workflows, thereby solving the shimming problem for any well-typed workflow,
5. implement our automated shimming technique and present a case study to validate the proposed approach.

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 does not insert shims in the workflow design, but instead performs implicit shimming by dynamically injecting coercions during workflow execution. While this paper focuses on primitive types, such as Int and Float, our general approach equally applies to structured data types as we explain in Section 6.
II. 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 \(dp_k; T_k\), where \(dp_k\) is the identifier, and \(T_k\) is the type of the data product.

Given a workflow \(W\) and the set of its constituent workflows \(W^*\), we use \(W_p\) to denote port \(p_i\) 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,

\[
W^*.IP = \{ip_i \mid ip_i \in W.IP, W_j \in W^*\}
\]
\[
W^*.OP = \{op_j \mid op_j \in W.OP, W_j \in W^*\}
\]

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. \(OP = \{op_0, op_1, ..., op_n\}\) is an ordered set of output ports,
4. \(W^* = \{W_0, W_1, ..., W_x\}\) is a set of constituent workflows used in \(W\). Each \(W_i \in W^*\) is another 9-tuple,
5. \(DP = \{dp_0, dp_1, ..., dp_n\}\) is a set of data products,
6. \(DC_{in} : IP \to W^* \times W^*.IP\) is an inverse-functional one-to-many mapping. \(DC_{in}\) is a set of ordered pairs:

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

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

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

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

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

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

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

That is, each pair in \(DC_{mid}\) represents a data channel connecting an output port \(op_k\) of some component \(W_j \in W^*\) with an input port \(ip_k\) of some other component \(W_k \in W^*\).

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

\[
DC_{idp} \subseteq \{(dp_k, (W_j, ip_k)) \mid dp_k \in DP, W_j \in W^*, ip_k \in W_j.IP\}
\]

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

Fig. 1 shows seven workflows that we will reference in this paper as \(W_a, W_b, W_c, W_d, W_e, W_f, W_g\) 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 port) or left (output port). Finally, data products are visualized as yellow boxes with their values placed inside (e.g., “true” in \(W_b\) in Fig. 1).

Because the order of input arguments of a workflow matters (e.g., Divide workflow in \(W_j\) in Fig. 1), we use ordered set IP to store a list of input ports. We use the term data channel to refer to any entity from the set \(DC_{in} \cup DC_{mid} \cup \cup DC_{out} \cup DC_{idp}\).

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. 1 as a 9-tuple, where $id = "W_d"$, $IP = \emptyset$, $OP = \{op_0, \text{Float}\}$, $W^* = \{\text{Mean, Sqrt}\}$, $DP = \{(dp_0, 3, \text{Int}), (dp_1, 5, \text{Int}), (dp_2, 4, \text{Int})\}$, $DC_{in} = DC_{out} = \emptyset$, $DC_{mid} = \{(\text{Mean, op}_0, \text{Sqrt}, (\text{op}_1, \text{ip}_2))\}$, $DC_{idp} = \{(\text{dp}_0, \text{Mean}, \text{ip}_3, \text{dp}_2, (\text{Mean, ip}_3))\}$, $W_{c} = \{\text{Mean, Sqrt}\}$, $DP_{in} = \emptyset$.

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 $W.IP \neq \emptyset \land W.OP \neq \emptyset \land W.W^* = W.DP = W.DC_{in} = W.DC_{mid} = W.DC_{out} = \emptyset$.

We use isPrimitive($W$) to denote the above predicate.

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 $(W.W^* \neq \emptyset \land W.IP \neq \emptyset \land W.OP \neq \emptyset \land W.DC_{in} = \emptyset \land W.DC_{idp} = \emptyset) \lor (W.W^* \neq \emptyset \land W.OP \neq \emptyset \land W.DP \neq \emptyset \land W.DC_{mid} \neq \emptyset)$.

We use isComposite($W$) to denote the above predicate.

All workflows $W_a, W_b, \ldots, W_g$ in Fig. 1 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 $W_a, W_b$, and $W_c$ in Fig. 1 are reusable. Workflow $W_d$ is reused inside $W_e$. Executable workflows, on the other hand have all input data needed to perform computation. Workflows $W_e, W_f, W_g$, and $W_i$ in Fig. 1 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).

III. WORKFLOW EXPRESSIONS

We rely on simply typed lambda calculus [2] 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 argument of type $\text{Int}$, 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, and if $W$ is executable, expr is an application.

In this paper, we present our 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 displayed horizontally with data flowing from left to right (see Fig. 1). Given a workflow $W$, our 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.

We accommodate composite workflows (containing sub-workflows) nested inside each other to arbitrary degree by making 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 our 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 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. 1. When the function translateWorkflow($W_d$) is called, it first checks whether $W_d$ is primitive, and because it is not, the else clause is executed (lines 7-35). translateWorkflow first determines that the component producing final result of the entire workflow $W_d$ is $\text{Sqrt}$ and stores it in the componentProducingFinalRes variable (line 16). Next, because $\text{Sqrt}$ has a single input, for loop in lines 21-23 executes once, calling the function getInputExpression($W_d, \text{Sqrt}, \text{ip}_2$), whose output “(Mean $dp_0, dp_1, dp_2$)” is stored into a string listOfArguments. Next, translateWorkflow checks whether $W_d$ is reusable (line 24), and because it is not it returns the application of workflow expression for $\text{Sqrt}$ component to the list of arguments obtained earlier (line 34). Since $\text{Sqrt}$ is a primitive workflow, translateWorkflow($\text{Sqrt}$) returns its name “$\text{Sqrt}$”. Thus, the final result of the translation is “$\text{Sqrt} \left( \lambda x_0 : \text{Int}. \text{Increment } x_0 \right)$”.

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

\begin{align*}
W_a & : \text{Increment } (\text{Not } dp_0) \\
W_b & : \lambda x_0 : \text{Bool}. \text{Increment } (\text{Not } x_0) \\
W_c & : (\text{Mean } dp_0, dp_1, dp_2) \\
W_d & : \text{Sqrt} \left( \lambda x_0 : \text{Int}. \lambda x_2 : \text{Int}. \lambda x_3 : \text{Int}. (\text{Sqrt} \left( \text{Mean } x_0, x_1, x_2 \right)) \right) \\
W_e & : \text{Divide} \left( \text{Increment} \left( \text{Square } dp_0 \right) \right) \left( \text{Decrement} \left( \text{Square } dp_0 \right) \right) \\
W_f & : \lambda x_0 : \text{Int}. \text{Divide} \left( \text{Increment} \left( \text{Square } x_0 \right) \right) \left( \text{Decrement} \left( \text{Square } x_0 \right) \right)
\end{align*}
Note that executable workflows \((W_a, W_c, W_d, W_j)\) are translated into lambda abstractions, whereas reusable ones \((W_b, W_e, W_g)\) into lambda abstractions. Ports are translated into variables, e.g. port \(i_p\) 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 \(W_c\) above). Thus, a workflow expression only contains port variables, names of primitive workflows, and data products.

### IV. Type System for Scientific Workflows

For interoperability, we adopt the type system defined in the XML Schema language specification [1]. While our approach can accommodate all types defined in [1], due to the space limit, in this paper we focus on the following types, which are most relevant to the scientific workflow domain:

- **String**
- **Decimal**
- **Integer**
- **NonPositiveInteger**
- **NegativeInteger**
- **NonNegativeInteger**
- **UnsignedLong**
- **UnsignedInt**
- **UnsignedShort**
- **UnsignedByte**
- **Double**
- **Float**
- **Long**
- **Int**
- **Short**
- **Byte**
- **Bool**

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 \(Bool \rightarrow Int\), because it expects boolean value as input and produces integer value as output. Workflow \(W_a\) has the type \(Int \rightarrow Int \rightarrow Double\). The type of an executable workflow is simply the type of its output, e.g., type of \(W_a\) is \(Int\).

We now introduce the notion of subtyping which is based on the fact that some types are more descriptive than others. For example, any value described by type \(Int\) can also be described by \(Decimal\). That is, the set of values associated with the type \(Int\) is a subset of values associated with the type \(Decimal\), or, in other words, \(Decimal\) is a more descriptive type than \(Int\). Therefore, it is safe to pass integer arguments to a workflow expecting a decimal number as input. Such view of subtyping, based on the subset semantics, is also called the principle of safe substitution. Workflows \(W_a, W_b,\) and \(W_c\) in Fig. 1 are composed by this principle.

We formalize the subtype relation as a set of inference rules used to derive statements of the form \(T_i \triangleleft T_j\), pronounced “\(T_i\) is a subtype of \(T_j\)”, or “\(T_j\) is a supertype of \(T_i\)”.
\[
\begin{align*}
\text{T} &<: \text{T} \quad \text{(S-REFL)} \\
\text{T}_1 <: \text{T}_2 &<: \text{T}_3 \quad \text{(S-TRANS)} \\
\{l_i: \text{T}_i \in 1..n\} &<: \{l_i: \text{T}_i \in 1..n\} \quad \text{(S-R)}
\end{align*}
\]

Fig. 2. Subtyping inference rules.

\(T_i\), or “\(T_i\) subsumes \(T_j\)”, where \(T_i\) and \(T_j\) are two types. As shown in Fig. 2, the first two rules (S-REFL, and S-TRANS) state that the subtype relation is reflexive and transitive. They are then followed by an inductive set of rules for primitive data types (collectively labeled S-Prim) derived from the hierarchy presented in [1]. As \(\text{Bool}\) type is less descriptive than \(\text{Int}\) (true and false can be mapped to 1 and 0, a subset of \(\text{Int}\)), we consider \(\text{Bool}\) to be a subtype of \(\text{Int}\). We also include the rule for records (S-R), which is a structured type. A record is a labeled n-tuple that has a type, e.g., \(r_i = \{a:1, \text{b:true, c: 2.25}\}\) is a 3-fields record, whose type is \(t_i = \{a: \text{Int}, b: \text{Bool}, c: \text{Float}\}\). We denote an n-fields record and its type as \(\{l_i: \text{T}_i\}, i \in 1..n\) where \(l_i, \text{v}_i\) and \(\text{T}_i\) denote labels, values and types respectively. Given two record types \(\text{T}_1\) and \(\text{T}_2\), \(\text{T}_1\) is subtype of \(\text{T}_2\) if all \(\text{T}_1\)’s fields form a superset of \(\text{T}_2\)’s fields. Indeed, it is safe to pass a record \(r_i\) where a record of type \(\{a: \text{Int}, b: \text{Bool}\}\) is expected, since \(r_i\) provides all necessary information (and even some extra) needed by the workflow.

**Definition 4.1 (Subtype relation).** A subtype relation is a binary relation between types, \(T_i <: T_j\) that satisfies all instances of the inference rules in Fig. 2.

Due to the small number of primitive types, the algorithm to check whether \(T_i <: T_j\) is true straightforward. We assume the function \(\text{subtype}(T_i, T_j)\) that returns true iff \(T_i <: T_j\).

V. Typechecking for 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 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 expression \((\text{Increment } dp\_0)\) has the type \(\text{Int}\), assuming \(\text{Increment}\) expects integer argument and returns integer (formally, \(\text{Increment: Int} \rightarrow \text{Int}\)) and \(dp\_0\) is \(\text{Int}\). In other words, we can derive workflow type given a set of assumptions.

Type derivation is done according to the following inference rules (see Fig. 3) for variables (T-VAR), abstractions (T-ABS), records (T-R), and applications (T-APP), as well as the rule for 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 [2], although modified to suit the scientific workflow domain and to ensure determinism of the typechecking algorithm presented later in this section.

Here, variable \(x\) represents a primitive object such as primitive workflow, port or data product, \(t, t\_\text{arg}\) and \(t\_\text{r}\) are lambda expressions, and \(T, T_i, T_j, T_{\text{in}}, T_{\text{out}}\) denote types. Set \(\Gamma = \{x_0: T_{p0}, x_1: T_{p1}, \ldots, x_n: T_{pn}\}\) is a 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 \(T_1 \rightarrow T_2\). 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 with subtyping connections (shown dashed in Fig. 1). We call such compositions workflows with subtyping. The last rule is used to typecheck records. We show 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 \(T\) in \(\Gamma\) there exists a typing derivation that satisfies all the inference rules in Fig. 3, and whose conclusion is \(Z + W : T\), where \(Z\) is a workflow context for \(W\).

**Example 5.3 (Typing derivation for workflow \(W\)).** Consider the workflow \(W\) shown in Fig. 1. Its workflow expression is \((\text{Increment } dp\_0)\). \(W\)’s workflow context \(Z\) is a set \(\{\text{Increment: Int} \rightarrow \text{Int}, \text{Not: Bool} \rightarrow \text{Bool}, dp\_0: \text{Bool}\}\). Typing derivation tree for this workflow is shown in Fig. 4. Each step is labeled with the corresponding inference rule. Derivation holds for \(\Gamma = Z\). According to Definition 5.2, existence of typing derivation with the conclusion \(\{\text{Increment: Int} \rightarrow \text{Int}, \text{Not: Bool} \rightarrow \text{Bool}, dp\_0: \text{Bool}\} \vdash \text{Increment (Not dp\_0): Int}\), proves that \(W\) is well-typed.

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 expression. 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. 3 by stating it “from bottom to top”. Note that for T-ABS we add variable-type pair for name \(x\), which is given explicitly in the abstraction.
Lemma 5.4 (Generation lemma).
1. \( \Gamma \vdash x : T \Rightarrow x : T \in \Gamma \) inverse T-VAR */
2. \( \Gamma \vdash (x : T_i, t) : T \Rightarrow \exists T_1 \exists T_2 \) \( (T = T_1 \land T_2 \land (\Gamma \cup \{x : T_i\} \cup \{t : T_j\})) */ inverse T-ABS */
3. \( \Gamma \vdash t_{arg} : T \Rightarrow \exists T_{in} \exists T_{out} \) \((\Gamma \vdash \cdot t_{in} \land \cdot T_{in}) \land (\Gamma \vdash \cdot t_{arg} \land \cdot T_{arg} \land \cdot (x : T_i) \land T_{j} : T_{j} \land T_{j} : T_{j})) */ inverse T-APP and T-APPS */

Proof: Part I - by contradiction. Assume \( \Gamma \vdash x : T \), and \( x : T \not\in \Gamma \). Since \( \Gamma \vdash x : T \), there must be a typing derivation satisfying inference rules in Fig. 3 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 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 \( \Gamma \vdash x : T \not\in \Gamma \) cannot be true, which is a contradiction. Parts 2 and 3 can be proved similarly by contradiction.

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 3. Given a workflow \( W \), it derives \( W \)'s type from

Algorithm 3. 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 */ i.e. variable representing port, primitive workflow or data product*/ then
5. return \( \Gamma \).getBinding(expr)
6. else if expr is Abstraction then
7. let \( \Gamma \) = \( \Gamma \)
8. \( \Gamma \)':addBinding(expr.name, exp.nameType)
9. typeOfExpr = typecheckWorkflow(expr.expression, \( \Gamma \)')
10. return expr.nameType = typeOfExpr
11. else if expr is Application then
12. typeOfE = typecheckWorkflow(expr.f, \( \Gamma \))
13. typeOfN = typecheckWorkflow(expr.n, \( \Gamma \))
14. if typeOfN is of the form \( T_n \rightarrow \cdots \rightarrow T_n \) then
15. if subtype(typeOfN, \( T_0 \)) then return \( T_n \rightarrow \cdots \rightarrow T_n \)
16. else
17. return "error: parameter type mismatch"
18. end if
19. else
20. return "arrow type expected"
21. end if
22. end if
23. end if
24. end function

the primitive objects inside \( W \) according to the typing rules in Fig. 3. This function is a transcription of the generation lemma (Lemma 5.4) that performs backward reasoning on the inference rules. Each recursive call of typecheckWorkflow is made according to the corresponding entry 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, \( \text{abstraction.name}, \text{abstraction.nameType} \) and \( \text{abstraction.expression} \) return name, type of name variable and expression of the given abstraction respectively.

VI. AUTOMATIC COERCION IN WORKFLOWS

Our approach not only allows to determine workflow well-typedness, but also ensures the correct execution of well-typed workflows. Consider the workflow \( W \) shown in Fig. 1. Although the \( \text{Bool} \) type is a subtype of \( \text{Int} \), 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 wrap data products of types \( \text{Bool} \) and \( \text{Int} \). 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.

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 \( \text{translateT} \) that translates workflow expressions with subtyping into those without subtyping. In this paper, we use \( C :: T_i \leftarrow T_j \) to denote subtyping derivation tree whose conclusion is \( T_i \leftarrow T_j \). Similarly, we use \( D :: \Gamma \vdash T \) to denote typing derivation whose conclusion is \( \Gamma \vdash T \). Given a subtyping derivation \( C :: T_i \leftarrow T_j \), function \( \text{translateC}(C) \) returns a coercion (lambda expression) that converts data products of type \( T_i \) into data products of type \( T_j \). We denote function \( \text{translateC}(C) \) as \([|C|]\) and define it in a case-by-case form:

\[
\begin{align*}
\text{translateT}([T_i < T_j]) &= \lambda x: T_i. x \\
\text{translateT}([\text{Bool} < \text{Int}]) &= \text{Bool2Int} \\
\text{translateT}([\text{Int} < \text{Long}]) &= \text{Int2Long} \\
\text{translateT}([T_i < T_j]) &= \text{translateT}(T_i, T_j) \\
\end{align*}
\]

The last case defines a function producing a record by extracting a subset of fields \((1..n)\) from an input record. Given a typing derivation \( D :: \Gamma \vdash T \), function \( \text{translateT}(D) \) produces an expression similar to \( T \) but in which subtyping is replaced with coercions. We also denote \( \text{translateT}(D) \) as \([|D|]\). From the context, it will be clear which one is used. Similarly, we define \( \text{translateT} \) by cases:

\[
\begin{align*}
\text{translateT}([x:T_i \in \Gamma]) &= x \\
\end{align*}
\]
Replace subtyping workflow - appropriate coercion and insert it, and.

Increment to -.

The XML structure is stored in a relational schema that coercion translates typing derivation for records by recursively calling itself on typing derivations of individual fields.

Example 6.1 (Automatic coercion injection). Consider the workflow $W_a$ in Fig. 1. To inject coercions into it, we call function $translateT$ on its typing derivation shown in Fig. 4. The function evaluates as follows

$$D_1 :: \Gamma \vdash \text{TranslateS} \Gamma \vdash \text{TranslateS} C : C \downarrow \Gamma \downarrow \text{Int}$$

The resulting expression contains coercion $\text{Bool2Int}$ that converts boolean data products to integer data products. Note that coercion $\text{Bool2Int}$ (implemented as a primitive workflow) is inserted dynamically at runtime and is transparent to the user.

VII. IMPLEMENTATION AND CASE STUDY

We now present the new version of our VIEW system [19], in which we implement our proposed workflow model, algorithms 1, 2, and 3, simply typed lambda calculus, and our translation functions $translateS$ and $translateT$. We give a walk-through explanation of our automated shimming technique using workflow $W_a$ in Fig. 1.

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 stored in a specification document written in our XML-based workflow language SWL. Fig. 5 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). The composed workflow is executed by pressing the “Run” button in the browser. First, using algorithms 1 and 2, our system translates workflow into typed lambda calculus with subtyping (bottom left section of the dialog in Fig. 5), and typechecks it using Algorithm 3. If the workflow is well-typed, using $translateS$ and $translateT$ functions, VIEW inserts coercions (primitive workflows performing type conversion) into the workflow expression by translating it into lambda calculus without subtyping. For example, coercion $\text{Bool2Int}$ is inserted in the expression for $W_a$ workflow (bottom right part of the dialog). Finally, the latter expression is translated back into a runtime version of SWL, which has necessary shims in it and is supplied to the workflow engine for execution.

Note that all these steps are fully automated and hidden from the user, who sees results of workflow execution upon pressing the “Run” button.

RELATED WORK

The significance of the shimming problem has been widely recognized by the scientific workflow community [3-8]. Much work to address shimming problem was focused on transforming XML documents whose elements are associated with domain models, (e.g., expressed using OWL) [10-12]. The common limitations of these approaches are: (1) they all focus on translating syntactically different XML documents, whereas other data types, including primitive, or structured types (e.g., record, relational schema) are not supported, (2) they all require services to be semantically annotated and hence they cannot compose arbitrary (not annotated) web services, let alone other kinds of executable components (e.g., scripts, local applications or HPC jobs).

Sellami et al. [9] 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. Ambite and Kapoor [13] present a planning approach to the
shimming problem that focuses on relational data types and
does not apply to primitive types or other non-relational
structured types (e.g., record). Existing scientific workflow
systems [14, 16, 17, 22] provide limited support to the
shimming problem, i.e. shimming is 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 [4], we presented a primitive
workflow model and a workflow specification language
that allowed hiding shims inside task specifications. This paper
removes 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
[2] to represent workflows which is naturally suitable for
dataflow modeling due to its functional characteristics [7].
While recognizing the importance of shims, [7] does not
address the shimming problem. We formalize coercion in
scientific workflows with type-theoretic rigor [2, 15].
Existing typechecking techniques apply in contexts other
than scientific workflows, e.g., Hindley-Milner algorithm
[21] requires typed prefix to typecheck expressions with
polymorphic types (not used in our model) 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 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.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we first reduced the shimming problem to
the runtime coercion problem from in theory of type
systems. Secondly, we proposed a scientific workflow
model and defined the notion of well-typed workflow.
Thirdly, we developed three algorithms to typecheck
workflows by first translating them into equivalent lambda
expressions. Fourthly, we designed two functions that
together insert “invisible shims”, or runtime coercions in
workflows, 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 a case study to
validate the proposed approach. In the future, we plan to
extend our technique to mediate structured data types such
as relational schema, and to develop real-world scientific
workflows relying on our implicit shimming approach.