A Theory of Typed Coercions and its Applications

Nikhil Swamy
Microsoft Research, USA
nswamy@microsoft.com

Michael Hicks [*]
University of Maryland, College Park
mwh@cs.umd.edu

Gavin M. Bierman
Microsoft Research Cambridge, UK
gmb@microsoft.com

Abstract

A number of important program rewriting scenarios can be recast as type-directed coercion insertion. These range from more theoretical applications such as coercive subtyping and supporting overloading in type theories, to more practical applications such as integrating static and dynamically typed code using gradual typing, and inlining code to enforce security policies such as access control and provenance tracking. In this paper we give a general theory of type-directed coercion insertion. We specifically explore the inherent tradeoff between expressiveness and ambiguity—the more powerful the strategy for generating coercions, the greater the possibility of several, semantically distinct rewritings for a given program. We consider increasingly powerful coercion generation strategies, work out example applications supported by the increased power (including those mentioned above), and identify the inherent ambiguity problems of each setting, along with various techniques to tame the ambiguities.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics; D.3.4 [*Processors*]: Compilers

General Terms Design, Languages, Theory

Keywords coercion insertion, nonambiguity, type-directed translation, provenance, gradual typing

1. Introduction

For nearly two decades, researchers have considered the problem of *type-directed coercion insertion*, in which data of one type can be automatically coerced to another type, without explicit intervention by the programmer [Breazu-Tannen et al. 1991, Barthe 1996, Luo 1996]. For example, suppose a value of type *lazy* α, called a *thunk*, represents a suspended computation that when evaluated will have type α. When needed, a thunk's underlying value is acquired by passing it to the function *force* : *lazy* $\alpha \rightarrow \alpha$. Rather than require the programmer to manually insert calls to *force*, the programmer can use *lazy* α values as if they were α values, and coercion insertion will add the needed calls to *force* automatically. Coercion insertion has been proposed to support numeric and other representational conversions, both in mainstream programming languages and type theories, and overloading, among other applications.

[*] Work performed while this author was visiting Microsoft Research, Cambridge, and the University of Cambridge Computer Laboratory.

ICFP'09, August 31–September 2, 2009, Edinburgh, Scotland, UK.
Copyright © 2009 ACM 978-1-60558-332-7/09/08...$10.00

We became interested in the coercion insertion problem while developing the Fable calculus [Swamy et al. 2008]. In that work we showed how to precisely enforce a broad range of security policies, including access control and noninterference-style information flow policies, by mediating access to security-relevant objects using what amounts to sophisticated coercions. For fine-grained, end-to-end security policies like information flow tracking, we found that manually inserting the necessary calls was tedious and resulted in confusing and complicated code. Thus it seemed natural to apply coercion insertion to add security checks automatically. Unfortunately, we found existing work on coercion insertion suffered from being too inexpressive or unpredictable because of ambiguous rewriting.

In this paper, we develop a new theory of type-directed coercion insertion for the simply-typed lambda calculus that consists of two interrelated judgments:

Coercion generation:	$\Sigma \vdash t \lhd_d t' \rightsquigarrow c$
Type-directed coercion insertion:	$\Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow m : t$

Both judgments are parameterized by a *coercion set* Σ consisting of the name and type of each primitive coercion, e.g., *force*:$\forall\alpha.$*lazy* $\alpha \rightarrow \alpha$.

The coercion generation judgment is used to construct coercion terms c of type $t \rightarrow t'$ built from primitive coercions in Σ. We study several definitions of the coercion generation judgment, each identified by an index d. The most powerful definition permits primitive coercions to have polymorphic types (though source terms do not), and may construct coercion terms by composing primitive coercions transitively (e.g., $\Sigma \vdash t_1 \lhd_d t_3 \rightsquigarrow c$ via $c_1 : t_1 \rightarrow t_2$ and $c_2 : t_2 \rightarrow t_3$) and component-wise according to type structure, as in coercive subtyping (e.g., $\Sigma \vdash t_1 \times t_2 \lhd_d t_1' \times t_2' \rightsquigarrow c$ via $c_1 : t_1 \rightarrow t_1'$ and $c_2 : t_2 \rightarrow t_2'$). No prior system has incorporated all three of these elements into coercion generation with the same degree of flexibility; e.g., Luo allows only primitive, polymorphic coercions [Luo and Kießling 2004, Luo 2008]; and while the coercions of Saïbi [1997] include elements of our structural and polymorphic judgments, his restrictions on the use of polymorphism make them unsuitable for our applications. As we demonstrate in §6.3, we have found the combination of the three to be crucial for implementing coercions that track information provenance [Cheney et al. 2007] whereby our framework inserts calls to these coercions automatically.

On the other hand, we show that weaker definitions of coercion generation are useful for avoiding potentially ambiguous rewritings, and may be preferred for some applications. With different coercion generation definitions we show our system can implement many applications, including overloaded operators (§2), equality-witnessing coercions (used in dynamic software update safety checking [Stoyle et al. 2007]) (§4.3), coercive subtyping [Luo 1996] (§5.3), forms of dynamic and gradual typing [Henglein 1994, Siek and Taha 2006] (§5.5), and transparent proxies for futures [Pratikakis et al. 2004] (§6.1). In several cases, we are the first to demonstrate that coercion insertion can be used to implement these applications.

The coercion insertion judgment $\Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow m : t$ is used to rewrite a term e by inserting generated coercions; e.g., to rewrite a source term $f\ 1$ to $(\mathit{force}\ f)\ 1$. The indices d, d' identify the coercion generation definition(s) to be used. For a given Σ there may be many ways to rewrite a term, in which case we say that coercion insertion is *ambiguous*. Ambiguity is a problem when different rewritings may have different run-time semantics, since programmers can no longer reason about the meaning of their source program. We find treatment of ambiguous rewriting in prior work lacking in two ways. First, rather than avoid ambiguity, several systems attempt to argue that all possible rewritings are confluent, and thus semantically equivalent. However, not all useful coercions exhibit the necessary equivalences to make such arguments. For example, coercions that cause a loss in precision may not have inverses, e.g., $(\mathit{int2float} \circ \mathit{float2int})\ x \neq x$, for all x. Additionally, when coercions and/or the source language include effects, such as I/O or run-time failure, different rewritings can be distinguished based on the order of evaluation. Second, prior work often does not consider the root causes of ambiguity due to Σ; for example, a Σ containing two coercions of type $t \rightarrow t'$ is problematic because any rewriting that uses the first coercion could just as well use the second. Work that does consider this problem is overly restrictive.

Our work addresses both of these limitations. To address the first problem, we employ a purely *syntactic* notion of ambiguity. We show that our syntactic restrictions make our system no less expressive than a system without such restrictions (which would instead rely on more limited confluence arguments).

For the second problem, we develop necessary and sufficient conditions that can be used to check, for a given Σ and coercion generation judgment, whether coercion insertion will be *strongly unambiguous*, meaning that every possible term rewriting is unambiguous (i.e., there is exactly one rewriting). Saïbi [1997] and Luo and Kießling [2004] also develop sufficient conditions, but they are overly strong, ruling out useful rewritings that would actually be unambiguous. Interestingly, we have found some useful coercion sets Σ that are *not* strongly unambiguous for some definitions of coercion generation—there will be ambiguous rewritings for at least some terms e. For these cases, the programmer must indicate via annotations which rewriting is preferred.

In summary, the primary contribution of this paper is a framework for type-directed coercion insertion that is extremely expressive, and yet carefully accounts for the problem of ambiguous rewritings (§2 and §3). We develop increasingly powerful definitions for coercion generation and use them to show how our framework can be used to encode many interesting applications (§4–§6), suggesting our approach to type-directed coercion insertion may be a worthwhile addition to mainstream programming languages.

2. Type-directed coercion insertion

We explore the type-directed coercion insertion problem for the simply typed lambda calculus; its syntax is given at the top of Figure 1. While we ultimately wish to support coercion insertion for a polymorphically-typed term language, simply-typed source terms nevertheless present significant challenges, and are relevant in many contexts, as we show throughout the paper. Types consist of base types B, function types $t_1 \rightarrow t_2$, and product types $t_1 \times t_2$, corresponding to the usual source language terms e. We also include type ascriptions $e{:}t$ and type casts $\langle t \rangle e$, which the programmer can use to direct the program rewriting and avoid ambiguity, as we discuss below.

Our coercion insertion judgment is written $\Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow m : t$. Here, Σ is a set of *primitive coercions* of the form $f{:}t_1 \rightarrow t_2$, where f is an identifier with type $t_1 \rightarrow t_2$. Primitive coercions are the building blocks used to rewrite source language terms e. The result of coercion insertion is a (type-correct) target language term m, paired with its type, to which e can be rewritten.

Coercions are produced by a *coercion generation* judgment, $\Sigma \vdash t \lhd_d t' \rightsquigarrow c$, a relation that states that a term of type t may be coerced to a term of type t' by applying coercion term c. An extremely simple definition of coercion generation, which we call *Base*, is shown in Figure 2. A coercion c is either drawn directly from Σ (CB-Prim) or is the identity coercion $id_t \equiv \lambda x{:}t.x$ (CB-Id). Sections 4, 5, and 6 consider progressively more powerful definitions of coercion generation, where each definition of $\Sigma \vdash t \lhd_d t' \rightsquigarrow c$ is distinguished by an index d (*Base* has index b). More powerful definitions enable more applications, but complicate reasoning about ambiguity. To tame such problems, we may use different definitions of coercion generation when rewriting particular sub-terms, as we explain below. As such, the coercion insertion judgment $\Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow m : t$ is parameterized by two indices, d and d', to indicate the forms of coercion generation to be used.

The inference rules for coercion insertion are shown at the middle of Figure 1. Rule (T-Var) simply produces the source variable x paired with its type; no coercion is applied. The rule (T-As) returns a rewritten term of the ascribed type t; later we illustrate how ascriptions can be used to locally resolve ambiguity. (T-Abs) and (T-Pair) simply rewrite the subterms as appropriate.

The rules (T-Cast), (T-Proj) and (T-App) capture the cases where a source term may have generated coercions inserted to produce a type-correct target term. In contrast to (T-As) which does not insert a coercion, (T-Cast) first rewrites the subterm e to a target term m of type t' and generates a coercion from t' to the required type t. If the generated coercion is the identity coercion, then we do not bother to insert it (this is captured using the $\langle\!\langle \cdot \rangle\!\rangle \cdot$ notation defined at the bottom of the figure). (T-Proj) rewrites the subterm e to a target term m of type t and inserts a generated coercion from t to a product type $t_1 \times t_2$. The (T-App) rule has the same basic form, except that both the functional component e_1 and the argument e_2 may have coercions applied.

These rules obey a design pattern: a source term is rewritten so that its type matches that required by the context in which it appears. If the context imposes no constraint, then no coercion is inserted. For example, (T-Var) simply returns x with its given type, rather than rewriting x by inserting a coercion from x's given type to some other type. Such eager coercion would result in no additional expressive power—the only time a coercion is needed for type-correctness is when x appears in an elimination position, as in $\mathsf{proj}_i(x)$ or $x\ e$ which requires x to have some type $t_1 \times t_2$ or $t_1 \rightarrow t_2$. We expect this basic pattern of inserting coercions at elimination positions should apply to many other standard programming language features, such as references, conditionals, and datatypes with pattern matching.

Note that, though not in elimination position, we also allow coercions to be applied to function arguments. The reason is that applying *structural* coercions, introduced in §5, to function terms may lead to ambiguity. Hence, we parameterize the coercion insertion judgment with two coercion generation forms, d and d', one for function (and pair) terms, one for function arguments.

2.1 Examples

Simple coercions. With the coercion set Σ and environment Γ defined as below,

$$\begin{array}{rcl} \Sigma & = & \mathit{ftoi}{:}\mathit{Float} \rightarrow \mathit{Int} \\ \Gamma & = & f{:}\mathit{Int} \rightarrow \mathit{Int},\ x{:}\mathit{Float} \end{array}$$

we can rewrite the term $f\ x$ using the *Base* coercion generation judgment according to $\Sigma; \Gamma \vdash_{b,b} f\ x \rightsquigarrow (f\ (\mathit{ftoi}\ x)) : \mathit{Int}$. This

Types	t, s	$::=$	$B \mid t_1 \to t_2 \mid t_1 \times t_2$
Source terms	e	$::=$	$x \mid e{:}t \mid \langle t \rangle e \mid \lambda x{:}t.e \mid e_1\ e_2 \mid (e_1, e_2) \mid \mathsf{proj}_i(e)$
Target terms	c, m	$::=$	$x \mid f \mid \lambda x{:}t.m \mid m_1\ m_2 \mid (m_1, m_2) \mid \mathsf{proj}_i(m)$

Primitive coercions	Σ	$::=$	$\cdot \mid \Sigma, f{:}t_1 \to t_2$
Typing context	Γ	$::=$	$\cdot \mid \Gamma, x{:}t$

$\boxed{\Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow m{:}t}$ Coercion generation def. $d, d' \ ::= \ b\,(\mathit{Base}) \mid r\,(\mathit{Trans}) \mid s\,(\mathit{Struct}) \mid p\,(\mathit{PolyTrans}) \mid q\,(\mathit{PolyStruct})$

$$\text{T-Var}\ \frac{}{\Sigma; \Gamma \vdash_{d,d'} x \rightsquigarrow x : \Gamma(x)} \qquad \text{T-As}\ \frac{\Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow m : t}{\Sigma; \Gamma \vdash_{d,d'} e{:}t \rightsquigarrow m : t} \qquad \text{T-Cast}\ \frac{\begin{array}{c}\Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow m : t' \\ \Sigma \vdash t' \lhd_{d'} t \rightsquigarrow c\end{array}}{\Sigma; \Gamma \vdash_{d,d'} \langle t \rangle e \rightsquigarrow \langle\!\langle c \rangle\!\rangle m : t}$$

$$\text{T-Abs}\ \frac{\Sigma; \Gamma, x{:}t \vdash_{d,d'} e \rightsquigarrow m : t'}{\Sigma; \Gamma \vdash_{d,d'} \lambda x{:}t.e \rightsquigarrow \lambda x{:}t.m : t \to t'} \qquad \text{T-Pair}\ \frac{\begin{array}{c}\Sigma; \Gamma \vdash_{d,d'} e_1 \rightsquigarrow m_1 : t_1 \\ \Sigma; \Gamma \vdash_{d,d'} e_2 \rightsquigarrow m_2 : t_2\end{array}}{\Sigma; \Gamma \vdash_{d,d'} (e_1, e_2) \rightsquigarrow (m_1, m_2) : t_1 \times t_2}$$

$$\text{T-Proj}\ \frac{\begin{array}{c}\Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow m : t \\ \Sigma \vdash t \lhd_d t_1 \times t_2 \rightsquigarrow c\end{array}}{\Sigma; \Gamma \vdash_{d,d'} \mathsf{proj}_i(e) \rightsquigarrow \mathsf{proj}_i(\langle\!\langle c \rangle\!\rangle m) : t_i} \qquad \text{T-App}\ \frac{\begin{array}{cc}\Sigma; \Gamma \vdash_{d,d'} e_1 \rightsquigarrow m_1 : t_1 & \Sigma \vdash t_1 \lhd_d t_1' \to t_2' \rightsquigarrow c \\ \Sigma; \Gamma \vdash_{d,d'} e_2 \rightsquigarrow m_2 : t_2 & \Sigma \vdash t_2 \lhd_{d'} t_1' \rightsquigarrow c'\end{array}}{\Sigma; \Gamma \vdash_{d,d'} e_1\ e_2 \rightsquigarrow (\langle\!\langle c \rangle\!\rangle m_1)\ (\langle\!\langle c' \rangle\!\rangle m_2) : t_2'}$$

$$\langle\!\langle c \rangle\!\rangle m \ = \ \begin{array}{ll} m & \text{if } c \equiv id_t \\ (c\ m) & \text{otherwise} \end{array}$$

Figure 1. Type-directed coercion insertion for the simply-typed lambda calculus

judgment states that applying f to x can be made type correct by leaving f alone and rewriting the argument x to be $\mathit{ftoi}\ x$. In fact, this is the only possible type-correct rewriting for this program.

Overloading. Luo [2008] has shown that coercions can be used to implement simple forms of overloading. Suppose we extend the Σ and Γ from above as follows:

$$\begin{array}{rcl} \Sigma' & = & \mathit{ptof}{:}P \to \mathit{Float} \to \mathit{Float} \to \mathit{Float}, \\ & & \mathit{ptoi}{:}P \to \mathit{Int} \to \mathit{Int} \to \mathit{Int}, \Sigma \\ \Gamma' & = & \mathit{plus}{:}P, \Gamma \end{array}$$

Here, Γ' includes a binary operator $\mathit{plus}{:}P$, where P is some distinguished base type, and the coercion set Σ' includes two coercions that allow plus to be applied both to a pair of Floats as well as a pair of Ints. We can prove $\Sigma'; \Gamma' \vdash_{b,b} \mathit{plus}\ x\ x \rightsquigarrow ((\mathit{ptof}\ \mathit{plus})\ x\ x) : \mathit{Float}$. This is perhaps the expected result, but is not the only possibility—another possible rewriting is $((\mathit{ptoi}\ \mathit{plus})\ (\mathit{ftoi}\ x)\ (\mathit{ftoi}\ x)) : \mathit{Int}$. It is not hard to see that each rewritten term has a different type, so the ambiguity can be resolved via a type ascription: source term $(\mathit{plus}\ x\ x){:}\mathit{Float}$ would produce the first and $(\mathit{plus}\ x\ x){:}\mathit{Int}$ would produce the second.

Note that $(\mathit{ftoi}\ ((\mathit{ptof}\ \mathit{plus})\ x\ x)) : \mathit{Int}$ is *not* a possible rewriting. As mentioned previously, this is because applying a coercion to the application itself has no impact on the type correctness of the rewritten term; i.e., $(\mathit{ptof}\ \mathit{plus})\ x\ x$ is already type-correct. The programmer could force this result using a type cast in the source term, writing it $\langle \mathit{Int} \rangle (\mathit{plus}\ x\ x)$.[1]

2.2 Type correctness

The goal of type-directed coercion insertion is to produce only type-correct target terms m that can be formed by rewriting a source term e. Additionally, we want to show that every well-typed source program e is also well-typed according to our coercion insertion judgment. We formalize these statements as follows, and

[1] Note that there are two rewritings for this term: $(\mathit{ftoi}\ ((\mathit{ptof}\ \mathit{plus})\ x\ x)) : \mathit{Int}$ and $((\mathit{ptoi}\ \mathit{plus})\ (\mathit{ftoi}\ x)\ (\mathit{ftoi}\ x)) : \mathit{Int}$. A type ascription can be used in the source term to further constrain the rewriting, i.e. $\langle \mathit{Int} \rangle ((\mathit{plus}\ \ x\ \ x){:}\mathit{Float})$ rewrites to $(\mathit{ftoi}\ ((\mathit{ptof}\ \mathit{plus})\ x\ x)) : \mathit{Int}$.

$$\text{CB-Prim}\ \frac{f{:}t \to t' \in \Sigma}{\Sigma \vdash t \lhd_b t' \rightsquigarrow f} \qquad \text{CB-Id}\ \frac{}{\Sigma \vdash t \lhd_b t \rightsquigarrow id_t}$$

Figure 2. Basic coercion generation (*Base*)

prove them for each coercion generation definition d developed subsequently: [2]

Theorem 1 (Soundness)**.**
$\forall \Sigma, \Gamma, e, m, t, d, d'. \Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow m : t \ \Rightarrow \Sigma, \Gamma \vdash m : t.$

Theorem 2 (Sufficiency)**.**
$\forall \Sigma, \Gamma, e, t, d, d'. \Gamma \vdash e : t \ \Rightarrow \ \Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow e : t.$

These propositions presume a standard type-checking judgment $\Gamma \vdash m : t$ on source and target terms. The context used to check the term m in the first judgment is Σ, Γ, the concatenation of the original primitive coercion list Σ and the original typing context Γ. The second proposition claims that when a source term e is typable, one possible rewriting of coercion insertion judgment includes the unmodified e.

3. Non-ambiguity

We would like to ensure that rewriting is *unambiguous*—there should be a single meaning attributable to e. We do this *syntactically*. Specifically, we state that a derivation is unambiguous when it produces at most one type-correct rewriting.

Definition 3 (Non-ambiguity)**.** *Given Σ, Γ, d, d', a term e can be* unambiguously *rewritten if there exists a unique m, t such that $\Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow m : t$.*

Our first example in §2.1 is unambiguous by this definition, while the second example is not.

[2] The proofs of all theorems for the fragment of our system in which $d, d' \in \{b, r, s\}$ are available as Coq proof scripts at `http://research.microsoft.com/~nswamy/papers/coercion-proofs.tgz`. Proofs of theorems for our polymorphic system can be found in a companion technical report [Swamy et al. 2009].

A syntactic notion of ambiguity has the benefit that we need not reason about the semantics of coercions, which is particularly useful in the presence of side effects. For example, suppose our *ftoi* coercion can sometimes fail, e.g., if its argument is *NaN*. Given the type-incorrect source term

$$\mathit{let}\ y = (\lambda z{:}\mathit{Int}.\mathit{NaN})\ 0\ \mathit{in}\ (e;\ 1 + y)$$

we could rewrite it to apply *ftoi* to either *NaN* within the body of the lambda term, or to y in the final expression $1 + y$. If e contains side effects, then the semantics of the two rewritten programs will be visibly different—the call to *ftoi* will fail prior to the execution of e in the first case, but after it in the second. Our system avoids this potential problem by preferring a single rewriting, and in this case produces only the second.

3.1 Strong non-ambiguity

Definition 3 considers non-ambiguity for a single term. We are also interested in non-ambiguity at the level of the rewriting system itself, i.e., for all terms. We refer to this property as strong non-ambiguity.

Definition 4 ($\mathsf{SNA}(\Sigma, d, d')$: Strong non-ambiguity)**.**

$$\begin{array}{l}\forall \Gamma, e, m_1, t_1, m_2, t_2.\\ \Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow m_1 : t_1 \wedge \Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow m_2 : t_2\\ \Rightarrow (m_1 = m_2 \wedge t_1 = t_2)\end{array}$$

Strong non-ambiguity depends on the particular forms of coercion generation and the set of primitive coercions Σ—for some definitions of coercion generation, and some Σ, coercion insertion will be strongly unambiguous, but for others it will not. For the second example in §2.1, *Base* coercion generation (Figure 2) used with the given Σ' causes coercion insertion to *not* be strongly unambiguous, evidenced by the fact that our example term $\mathit{plus}\ x\ x$ has two possible rewritings. On the other hand, it turns out that with the Σ from the first example, *Base is* strongly unambiguous: there is no possibility of producing multiple rewritings for any program under $\Sigma = \mathit{ftoi}{:}\mathit{Float} \to \mathit{Int}$.

3.2 Establishing strong non-ambiguity

Strong non-ambiguity is a property of the program rewriting system, but we can prove that it can be characterized by three constraints (NAC_π, $\mathsf{NAC}_\times$, and NAC_{app}) on the coercion generation judgment, for a given Σ.

The most obvious constraint is that coercion generation must be a partial function from pairs of types to coercions—given some Σ, coercion generation should produce at most one term c that coerces values between given source and target types t and t'. In later sections, we show how this constraint can be decided by viewing a coercion term as a path in a graph. We can formalize this requirement as follows:

Definition 5 ($\mathsf{NAC}_\pi(d, \Sigma)$: unique coercion paths)**.**

$$\forall t, t', c, c'. \Sigma \vdash t \lhd_d t' \rightsquigarrow c \ \wedge\ \Sigma \vdash t \lhd_d t' \rightsquigarrow c' \ \Rightarrow\ c = c'$$

To guarantee strong non-ambiguity, we require two further constraints that arise from the way terms are rewritten by the coercion insertion judgment.

First, when rewriting a term $\mathsf{proj}_i(e)$ using the (T-Proj) rule, if there is more than one product type to which to coerce e, the rewriting could be ambiguous. For example, it could be that $\Sigma \vdash t \lhd_d t_1 \times t_2 \rightsquigarrow c$ and $\Sigma \vdash t \lhd_d t'_1 \times t'_2 \rightsquigarrow c'$, and as such (assuming e has type t), we could rewrite $\mathsf{proj}_i(e)$ to both $\mathsf{proj}_i(c\ e){:}t_i$ and $\mathsf{proj}_i(c'\ e){:}t'_i$. Thus we must require that any type t can be coerced to at most one product type $t_1 \times t_2$—we formalize this condition as follows:

$$\text{Coercion paths} \quad \pi \quad ::= \quad \{t_1, \dots, t_k\}$$

$$\text{CC-Id}\ \frac{}{\Sigma \vdash^{\pi} t \lhd_r t \rightsquigarrow id_t}$$

$$\text{CC-PrimTrans}\ \frac{\begin{array}{c} f{:}t \to t'' \in \Sigma \\ \Sigma \vdash^{\pi \uplus \{t''\}} t'' \lhd_r t' \rightsquigarrow c \end{array}}{\Sigma \vdash^{\pi} t \lhd_r t' \rightsquigarrow c \circ f}$$

$$\text{CC-InitPath}\ \frac{\Sigma \vdash^{\{t\}} t \lhd_r t' \rightsquigarrow c}{\Sigma \vdash t \lhd_r t' \rightsquigarrow c}$$

Figure 3. Transitive composition of coercions (*Trans*)

Definition 6 ($\mathsf{NAC}_\times(d, \Sigma)$: unique coercion to a product type)**.**

$$\begin{array}{l}\forall t, t_1, t_2, t'_1, t'_2, c, c'.\\ \quad \Sigma \vdash t \lhd_d (t_1 \times t_2) \rightsquigarrow c \ \wedge\ \Sigma \vdash t \lhd_d (t'_1 \times t'_2) \rightsquigarrow c' \ \Rightarrow\\ \qquad t_1 = t'_1 \wedge t_2 = t'_2\end{array}$$

A similar situation arises when rewriting a term $e_1\ e_2$ using the (T-App) rule. Assume that e_1 is rewritten to a term of type t_1 and e_2 is rewritten to a term of type t_2. To preserve strong non-ambiguity there must be only one way to coerce the type t_1 to a function type $t' \to t''$ (where t_2 can be coerced to t'). This property is defined as follows:

Definition 7 ($\mathsf{NAC}_{app}(d, d', \Sigma)$: unambiguous applications)**.**

$$\begin{array}{l}\forall t_1, t_2, t_3, t_4, t_5, t_6, c_1, c_2, c_3, c_4.\\ \quad \Sigma \vdash t_1 \lhd_d (t_3 \to t_5) \rightsquigarrow c_1 \quad \wedge \quad \Sigma \vdash t_2 \lhd_{d'} t_3 \rightsquigarrow c_3 \wedge\\ \quad \Sigma \vdash t_1 \lhd_d (t_4 \to t_6) \rightsquigarrow c_2 \quad \wedge \quad \Sigma \vdash t_2 \lhd_{d'} t_4 \rightsquigarrow c_4\\ \quad \Rightarrow\ (t_3 = t_4 \ \wedge\ t_5 = t_6)\end{array}$$

The theorem below establishes that these three conditions are both necessary and sufficient for strong non-ambiguity.

Theorem 8 (Strong non-ambiguity)**.**

$$\begin{array}{l}\forall \Sigma, d, d'. \mathit{SNA}(\Sigma, d, d') \ \Leftrightarrow\\ \mathit{NAC}_\pi(d, \Sigma) \wedge \mathit{NAC}_\pi(d', \Sigma) \wedge \mathit{NAC}_\times(d, \Sigma) \wedge \mathit{NAC}_{app}(d, d', \Sigma)\end{array}$$

Additional conditions similar to these may be needed to handle other language features. For example, a condition analogous to NAC_{app} would be needed to handle assignments to references $e_1 := e_2$. On the other hand, no additional condition is needed to support conditionals; NAC_π is sufficient.

It is easy to check that our overloading example $\Sigma = \mathit{ptof}{:}P \to \mathit{Float} \to \mathit{Float} \to \mathit{Float}, \mathit{ptoi}{:}P \to \mathit{Int} \to \mathit{Int} \to \mathit{Int}$ is strongly unambiguous. However, if the coercion $\mathit{ftoi}{:}\mathit{Float} \to \mathit{Int}$ is included in Σ then $\mathsf{NAC}_{app}(b, b, \Sigma)$ no longer holds, so ambiguous derivations become possible, as we have seen. In subsequent sections we present algorithms that either precisely decide or approximate these NAC constraints for particular coercion generation definitions.

4. Composing primitive coercions

We can increase the number of rewritable programs by increasing the power of the coercion generation judgment. However, as expressive power increases so, too, does the likelihood of ambiguous rewritings. In this and the next two sections we define more powerful coercion generation judgments that can generate *compound* coercions. In this section we consider *Trans*, which may generate compound coercions by transitive composition (e.g., $\Sigma \vdash t_1 \lhd_d t_3 \rightsquigarrow c$ via $c_1 : t_1 \to t_2$ and $c_2 : t_2 \to t_3$); in §5 we consider *Struct*, which extends *Trans* to generate compound coercions component-wise, according to type structure (e.g., $\Sigma \vdash t_1 \times t_2 \lhd_d t'_1 \times t'_2 \rightsquigarrow c$ via $c_1 : t_1 \to t'_1$ and $c_2 : t_2 \to t'_2$); and in §6 we consider *PolyTrans* and *PolyStruct*, which extend *Trans* and *Struct*, respectively, to support primitive coercions with polymorphic types. In each section we explore substantial applications enabled by this added power, and consider conditions that are sufficient to ensure strong non-ambiguity.

4.1 Generating composite coercions

A natural extension to the *Base* judgment is to allow the sequential composition of primitive coercions. We call this extended relation *Trans*; its rules are given in Figure 3. Coercion terms c consist of the identity coercion id_t and/or the composition of some number of primitive coercions, written $c \circ f$ (equivalent to $\lambda x.c\ (f\ x)$). Coercion generation $\Sigma \vdash t \lhd_r t' \rightsquigarrow c$ is given by a single top-level rule (CC-InitPath), which in turn appeals to the judgment $\Sigma \vdash^{\pi} t \lhd_r t' \rightsquigarrow c$, whose definition is rather straightforward. (We discuss the parameter π shortly.)

While the idea of generating composite coercions appears a simple extension, it adds potential for some subtle ambiguities. The rules in Figure 3 are designed to prevent two forms of ambiguity that we call *syntactic ambiguity* (SA) and *cyclical ambiguity* (CA). Fortunately, both can be avoided with no loss to expressive power.

Syntactic ambiguity would arise from having a general rule of transitivity instead of embedding it as in rule (CC-PrimTrans). For example, suppose we have $\Sigma = f{:}A \to B, g{:}B \to C$. Then to coerce A to C, *Trans* offers one possibility: $\Sigma \vdash A \lhd_r C \rightsquigarrow (id_C \circ g) \circ f$. However if we had extended *Base* with a general rule of transitivity:

$$\text{Strawman-Trans}\ \frac{\Sigma \vdash t \lhd_b t' \rightsquigarrow c \qquad \Sigma \vdash t' \lhd_b t'' \rightsquigarrow c'}{\Sigma \vdash t \lhd_b t'' \rightsquigarrow c' \circ c}$$

then we would be able to generate many different coercions: $\Sigma \vdash A \lhd_b C \rightsquigarrow (id_C \circ g) \circ f$, $\Sigma \vdash A \lhd_b C \rightsquigarrow id_c \circ (g \circ f)$, $\Sigma \vdash A \lhd_b C \rightsquigarrow (id_c \circ (id_c \circ g)) \circ f$, $\Sigma \vdash A \lhd_b C \rightsquigarrow (id_c \circ g) \circ (id_b \circ f)$, etc. While each of these coercions is semantically equivalent, and we could endeavor to prove that this is the case, we find it cleaner to force syntactic and semantic ambiguity to correspond, and then focus on eliminating the former. To this end, our rules force a left-associative composition of coercions, with exactly one application of the identity coercion at the end.[3]

As an example of cyclical ambiguity consider the coercion set $\Sigma = f{:}A \to B, b{:}B \to A$. Then in our strawman judgment we can prove both $\Sigma \vdash A \lhd_b A \rightsquigarrow id_A$ and $\Sigma \vdash A \lhd_b A \rightsquigarrow b \circ f$ (and $\Sigma \vdash A \lhd_b A \rightsquigarrow b \circ f \circ b \circ f$, ad infinitum). Clearly Σ is the source of the problem, as it contains coercions that can be composed in cycles. However, we cannot simply rule out Σ such as this one, since doing so would rule out useful applications (we give an example in §4.3). Instead, we disallow cyclic compositions by using a *path* parameter π, which records the domain type of each primitive coercion composed in sequence; rule (CC-InitPath) initializes the path with the original source type t. The (CC-PrimTrans) rule adds the intermediate type t'' to the path *if it is not already present*, since addition is via the disjoint union operator $\uplus$. Thus no composed coercion ever "visits" the same type twice.

4.2 Strong non-ambiguity

To use the *Trans* relation to rewrite terms, we write our coercion insertion judgment as $\Sigma; \Gamma \vdash_{r,r} e \rightsquigarrow m : t$. In order for rewriting to be strongly unambiguous, we need to check the NAC constraints of §3.2. One natural way to do so is to view Σ as a graph on types.

Definition 9 (Coercion Graph)**.** *A coercion set Σ induces a coercion graph $G_\Sigma = (V, E)$, where $(t, t') \in E$ iff $\exists f. \Sigma(f) = t \to t'$ and $V = \bigcup_{(t,t') \in E} \{t, t'\}$. We write $\mathsf{Reachable}(\Sigma, t, t')$ whenever a (possibly empty) sequence of edges from t leads to t' in G_Σ.*

[3] We could use a device similar to $\langle\!\langle c \rangle\!\rangle m$ from Figure 1 to eliminate the trailing id_t.

$$\begin{aligned}
\Sigma &= con_X{:}X \to Int,\ abs_X{:}Int \to X,\\
&\ con_Y{:}Y \to X,\ abs_Y{:}X \to Y\\
\Gamma &= y{:}Y,\ f{:}X \to X \to X\\
&\Sigma; \Gamma \vdash_{r,r} f\ y\ 1 \rightsquigarrow f\ (id_X \circ con_Y\ y)\ \ (id_X \circ abs_X\ 1)\ : \mathrm{X}
\end{aligned}$$

Figure 4. Proteus Abs/con example derivation

$\mathsf{NAC}_\pi(r, \Sigma)$ holds iff a depth-first search started from each node in the graph produces no cross or forward edges. Such edges indicate two potential paths, and thus two possible compositions of primitive coercions, from the source node's type to the type of the forward or cross edge's target node. Back edges are not problematic—they indicate the potential for cycles, which are ruled out by the judgment's π parameter. $\mathsf{NAC}_\times(r, \Sigma)$ can be decided by constructing the set $P_t = \{t_1 \times t_2 \mid \mathsf{Reachable}(\Sigma, t, t_1 \times t_2)\}$, for every node's type t in G_Σ, and checking $|P_t| \le 1$. To decide $\mathsf{NAC}_{app}(r, r, \Sigma)$, we compute the set $A_t = \{t_2 \mid \exists t_1, t'.\mathsf{Reachable}(\Sigma, t, t_1 \to t_2) \wedge \mathsf{Reachable}(\Sigma, t', t_1)\}$, for each node type $t \in G_\Sigma$, and check $|A_t| \le 1$. The construction of A_t and P_t can be interleaved with the depth-first traversals of G_Σ. Thus, the complexity of deciding strong non-ambiguity for the *Trans* judgment is $O(n^2)$ where $n = |\Sigma|$.

4.3 Application: Proteus abs/con insertion

An interesting example of the program rewriting enabled by *Trans* arises from dynamic software updating (DSU). DSU is a technique by which a running program can be updated with new code and data without interrupting its execution. A core problem in DSU arises from timing; particular updates are legal only at certain times.

In prior work [Stoyle et al. 2007] we defined a language Proteus which supported type equations for named types, e.g., $typename\ X = t$ where t is the definition of X (and may itself be, or contain, abstract type names). A named type's definition can be updated on-the-fly provided that no actively running function references the old representation. Proteus employs an explicit coercion con_X to witness the use of a value of type X as one of type t, while coercion abs_X witnesses the reverse. An update to type X is permitted so long as the text of actively running functions contains no mention of abs_X or con_X coercions.

As such explicit coercions are rather cumbersome, Proteus supports a custom algorithm that takes a source program with (abstract) type equations and automatically inserts the needed coercions. This coercion insertion algorithm can be expressed precisely using our coercion insertion framework. An example is shown in Figure 4. We can show that the algorithm is strongly unambiguous, by the proposition below.

Proposition 10 (Abs/con strongly unambiguous)**.** *Assuming a set of base types $X_1, X_2, ...X_n$, let $\Sigma = \bigcup_{1 \le i \le n} \{con_{X_i}{:}X_i \to t_i,\ abs_{X_i}{:}t_i \to X_i\}$. Then $\mathsf{SNA}(r, r, \Sigma)$.*

Equality coercions like abs/con are also useful in type-preserving compilation, inserted as proofs of term equality in the typed intermediate language [Sulzmann et al. 2007]. We conjecture our system could be used in this setting as well.

5. Structural coercions

This section presents *Struct*, a coercion generation definition that extends *Trans* to also produce lifted coercions over function and product type constructors; we call such synthesized coercions *structural coercions*. For example, given coercions $f{:}t_1 \rightarrow s_1$ and $g{:}t_2 \rightarrow s_2$, *Struct* can generate coercions from $t_1 \times t_2$ to $s_1 \times s_2$ and from $s_1 \rightarrow t_2$ to $t_1 \rightarrow s_2$. *Struct* adds further sources of potential ambiguity, and we employ several devices to eliminate them without unduly compromising expressiveness—we show that *Struct* is expressive enough to represent a canonical form of coercive subtyping. Furthermore, we define efficiently decidable, sufficient conditions for proving strong non-ambiguity. This section concludes with examples that use structural coercions to encode forms of dynamic and gradual typing [Henglein 1994, Siek and Taha 2006].

5.1 Coercion generation

The definition of *Struct* in Figure 5 begins with a top-level rule (CS-Init), which initializes two indices a and π for controlling ambiguity. After first explaining the general structure of the remaining rules we consider these ambiguity controls in detail.

The (CS-Id) and (CS-PrimTrans) rules are analogous to the versions of these rules in Figure 3. (CS-Id) generates the identity coercion for a type t, while (CS-PrimTrans) allows a primitive coercion f to be composed transitively with some (potentially complex) coercion c generated in the second premise.

The (CS-FunTrans) rule constructs a coercion from a function type $t_1 \rightarrow t_2$ to some target type t' by first assembling a coercion to another function type $t'_1 \rightarrow t'_2$, and then composing it with a coercion from $t'_1 \rightarrow t'_2$ to t'. As in structural subtyping, coercion generation on functions is contravariant in the argument and covariant in the return type. So, in the first premise of (CS-FunTrans), we construct a coercion c_1 from t'_1 to t_1, and, in the second premise, a coercion c_2 from t_2 to t'_2. The third premise defines a term c that, when applied to a function f of type $t_1 \rightarrow t_2$, will coerce it to type $t'_1 \rightarrow t'_2$ by waiting until f is eventually applied and, at that point, applying the coercions c_1 and c_2 to the arguments and return value of f, respectively. Note that both c_1 and c_2 are constructed by making use of the *Struct* relation. This allows us to use the structural rules to an arbitrary depth when coercing higher-order function types; e.g., with the appropriate coercion set Σ, (CS-FunTrans) allows us to generate coercions between types of the form $(t_1 \rightarrow t_2) \rightarrow t_3 \lhd_s (t'_1 \rightarrow t'_2) \rightarrow t'_3$.

The rule (CS-PairTrans) is similar to (CS-FunTrans) and provides structural coercions for product types. The coercions c_1 and c_2 are covariant on each component of the product type—again, these may themselves make use of the structural rules in order to construct coercions between nested products.

5.2 Eliminating cyclical and syntactic ambiguity

The *Trans* judgment was formulated to remove possible syntactic and cyclical ambiguities. Both forms of ambiguity are also present in the *Struct* relation and we employ similar devices to eliminate them. For cyclical ambiguity, the judgment is indexed by a path π that is initialized in (CS-Init), and, just as before, every rule (except (CS-Id)) checks and augments π.

To avoid syntactic ambiguity, the associativity of coercion composition is normalized in (CS-PrimTrans) as in *Trans*, and the same device is applied by (CS-FunTrans) and (CS-PairTrans) since these rules also employ transitivity.

Struct employs an additional index a on the turnstile to eliminate a form of syntactic ambiguity new to *Struct*. Without restriction it is possible to construct multiple coercions between the same constructed types by using different interleavings of structural coercion rules and transitivity. As an illustration, consider how we could coerce a product type $t_1 \times t_2$ to another product type $s_1 \times s_2$ using (CS-PairTrans) with $\Sigma = f{:}t_1 \rightarrow s_1, g{:}t_2 \rightarrow s_2$:

$$\text{CS-PairTrans}\ \frac{\begin{array}{c}\Sigma \vdash t_1 \lhd_s s_1 \rightsquigarrow f \qquad \Sigma \vdash t_2 \lhd_s s_2 \rightsquigarrow g \\ c = \lambda x{:}t_1 \times t_2.(f\ \mathsf{proj}_1(x), g\ \mathsf{proj}_2(x)) \\ \Sigma \vdash_{\ominus}^{\pi \uplus \{s_1 \times s_2\}} s_1 \times s_2 \lhd_s s_1 \times s_2 \rightsquigarrow id_{s_1 \times s_2}\end{array}}{\Sigma \vdash_{\oplus}^{\pi} t_1 \times t_2 \lhd_s s_1 \times s_2 \rightsquigarrow id_{s_1 \times s_2} \circ c}$$

Without the a index to prevent us, we could also construct the coercion thus:

$$(2)\ \frac{\begin{array}{c}\Sigma \vdash t_1 \lhd_s t_1 \rightsquigarrow id_{t_1} \qquad \Sigma \vdash t_2 \lhd_s s_2 \rightsquigarrow g \\ c = \lambda x{:}t_1 \times t_2.(\mathsf{proj}_1(x), g\ \mathsf{proj}_2(x)) \\ \Sigma \vdash^{\pi \uplus \{t_1 \times s_2\}} t_1 \times s_2 \lhd_s s_1 \times s_2 \rightsquigarrow c' \\ c' = id_{s_1 \times s_2} \circ (\lambda x{:}t_1 \times s_2.(f\ \mathsf{proj}_1(x), \mathsf{proj}_2(x)))\end{array}}{\Sigma \vdash^{\pi} t_1 \times t_2 \lhd_s s_1 \times s_2 \rightsquigarrow c' \circ c}$$

Here, c' is produced by another application of (CS-PairTrans). If f and g were effectful (suppose each printed a message) and we assume left-to-right evaluation order, then the first case, when the constructed coercion is evaluated f would print its message first, then g, while in the second case, g would print first, then f.

To avoid this issue, we observe that in many circumstances (characterized precisely by Theorem 11, below), if we can prove $\Sigma \vdash t \lhd_s t' \rightsquigarrow c$ by applying structural rules (CS-PairTrans) and/or (CS-FunTrans) in succession, as in the second case above, we can also prove $\Sigma \vdash t \lhd_s t' \rightsquigarrow c'$ via a derivation that uses a single structural rule followed by a non-structural rule, as in the first case. Therefore, we augment *Struct*'s rules with an index a to prevent structural rules from being used in succession. In particular, in the conclusion of (CS-PairTrans) and (CS-FunTrans) we require the index $a = \oplus$; in the last premise of these rules, we require the index $a = \ominus$. For our example above, this device forces the first rewriting. Non-structural rules may still occur in succession; (CS-PrimTrans) places no limit on the index of its second premise.

There is one final source of syntactic ambiguity which arises because the rules in Figure 1 permit inserting coercions on both the left and right-hand-side of applications. To illustrate, suppose we have $\Sigma = \mathit{ftoi} : \mathit{Float} \rightarrow \mathit{Int}$ and $\Gamma = f{:}\mathit{Int} \rightarrow \mathit{Int}, x{:}\mathit{Float}$. Then the term $(f\ \ x)$ can be rewritten to $(f\ \ (\mathit{ftoi}\ \ x))$ and to $((\lambda g{:}\mathit{Int} \rightarrow \mathit{Int}.\lambda y{:}\mathit{Float}.g\ (\mathit{ftoi}\ y))\ f)\ x)$. The choice depends on whether we rewrite the left-hand or right-hand-side of the application. In general, if we can rewrite a function's argument, as in the former case, we could have rewritten the function using (CS-FunTrans), as in the latter case.

To avoid this syntactic ambiguity, we may use the *Struct* rules on either the left- or right-hand-side of an application, but not both. We choose to use *Struct* only on the right-hand-side because applying it on the left-hand-side would create another problem. In particular, for some Σ the *Struct* rules can coerce a type t to infinitely many function types, all with the same domain type t'. For example, with $\Sigma = f{:}t \rightarrow (t' \rightarrow t)$, we can construct $t \lhd_s (t' \rightarrow t), t \lhd_s (t' \rightarrow (t' \rightarrow t), t \lhd_s (t' \rightarrow (t' \rightarrow (t' \rightarrow t)))$, etc. A similar problem can arise when rewriting a term $\mathsf{proj}_i(x)$—for certain Σ, there may be an infinite number of product types to which a given type can be coerced.

To control the choice of coercion generation during rewriting, we parameterize the rewriting judgment by two indices d and d', writing it $\Sigma; \Gamma \vdash_{d,d'} e \rightsquigarrow m : t$. Following the reasoning outlined above, we use the judgment $\Sigma \vdash_{r,s} e' \rightsquigarrow m : t$, which indicates that *Trans* is used to rewrite terms e that appear within a projection or on the left-hand-side of function applications, while *Struct* is used to rewrite terms on the right-hand-side of function applications. Consider an application $e\ e'$ where the subterm e is rewritten to a term of type t and e' is rewritten to type t'. By restricting to the use of *Trans*, there are finitely many function types that t can be coerced to. It is then decidable to use *Struct* to generate a coercion from t' to the domain of each function type.

Index on turnstile (alternation of structural rules) $\quad a \quad ::= \quad \ominus \mid \oplus$

$$\text{CS-Init}\ \frac{\Sigma \vdash_{\oplus}^{\{t\}} t \lhd_s t' \rightsquigarrow c}{\Sigma \vdash t \lhd_s t' \rightsquigarrow c} \qquad \text{CS-Id}\ \frac{}{\Sigma \vdash_a^{\pi} t \lhd_s t \rightsquigarrow id} \qquad \text{CS-PrimTrans}\ \frac{f{:}t \to t_1 \in \Sigma \qquad \Sigma \vdash_{a'}^{\pi \uplus \{t_1\}} t_1 \lhd_s t' \rightsquigarrow c}{\Sigma \vdash_a^{\pi} t \lhd_s t' \rightsquigarrow c \circ f}$$

$$\text{CS-FunTrans}\ \frac{\begin{array}{c}\Sigma \vdash t_1' \lhd_s t_1 \rightsquigarrow c_1 \qquad \Sigma \vdash t_2 \lhd_s t_2' \rightsquigarrow c_2 \\ c = \lambda f{:}t_1 \to t_2.(\lambda x{:}t_1'.c_2\ (f\ (c_1\ x))) \\ \Sigma \vdash_{\ominus}^{\pi \uplus \{t_1' \to t_2'\}} t_1' \to t_2' \lhd_s t' \rightsquigarrow c'\end{array}}{\Sigma \vdash_{\oplus}^{\pi} t_1 \to t_2 \lhd_s t' \rightsquigarrow c' \circ c} \qquad \text{CS-PairTrans}\ \frac{\begin{array}{c}\Sigma \vdash t_1 \lhd_s t_1' \rightsquigarrow c_1 \qquad \Sigma \vdash t_2 \lhd_s t_2' \rightsquigarrow c_2 \\ c = \lambda x{:}t_1 \times t_2.(c_1\ \mathsf{proj}_1(x), c_2\ \mathsf{proj}_2(x)) \\ \Sigma \vdash_{\ominus}^{\pi \uplus \{t_1' \times t_2'\}} t_1' \times t_2' \lhd_s t' \rightsquigarrow c'\end{array}}{\Sigma \vdash_{\oplus}^{\pi} t_1 \times t_2 \lhd_s t' \rightsquigarrow c' \circ c}$$

Figure 5. Generating structural coercions (*Struct*)

5.3 Expressiveness

While the additional devices outlined above eliminate problematic ambiguities for *Struct*, we may be concerned that they also unduly reduce its expressiveness. The theorem below establishes that this is not the case in many common situations. We compare our system to a standard declarative formulation of subtyping for the lambda calculus, in which subtyping can be applied to any term, and uses of transitivity are unrestricted. The rules are essentially standard, and we elide them. To simplify the proof, we write this judgment $\Sigma; \Gamma \vdash e : t$, where each coercion $f{:}t \to t' \in \Sigma$ is read as a subtyping relation $t <: t'$ between *base types*; typically subtyping between base types would be expressed as distinct rules. As is usual with subtyping systems, we do not consider base subtyping relations between structured types. Thus, for this theorem, we do not permit primitive coercions of the form $f{:}(t_1 \to t_2) \to t' \in \Sigma$ and denote this constraint on Σ as $BaseCoercions(\Sigma)$.

Theorem 11 (Expressiveness of structural coercions)**.**

$$\begin{array}{l}\forall \Sigma, \Gamma, e, t.\ BaseCoercions(\Sigma)\ \wedge\ \Sigma; \Gamma \vdash e : t \Rightarrow \\ \exists m, t', c.\ \Sigma; \Gamma \vdash_{r,s} e \rightsquigarrow m{:}t' \wedge \Sigma \vdash t' \lhd_s t \rightsquigarrow c\end{array}$$

This theorem establishes that our coercion insertion system is at least as expressive as a system that provides a standard form of structural subtyping. One interesting point to note is that our coercion insertion system produces a term typable at t', a *subtype* of the type t produced by the subtyping system. This is an artifact of the restriction in our system that inserts coercions only in elimination forms. Of course, a top-level coercion cast can always be used to produce a term of the desired super-type of t'.

5.4 Strong non-ambiguity

Proving strong non-ambiguity for the coercion insertion judgment $\Sigma; \Gamma \vdash_{r,s} e \rightsquigarrow m : t$ requires establishing $\mathsf{NAC}_{\times}(r, \Sigma)$, $\mathsf{NAC}_{app}(r, s, \Sigma)$, $\mathsf{NAC}_{\pi}(r, \Sigma)$, and $\mathsf{NAC}_{\pi}(s, \Sigma)$ defined in §3. It turns out to be particularly difficult to decide $\mathsf{NAC}_{\pi}(s, \Sigma)$ for arbitrary Σ. One difficulty (as mentioned in §5.2) is that using the *Struct* relation, one can coerce a type t to infinitely many other types. Furthermore, given a coercion set $\Sigma = f_1{:}t_1 \to t_1', \ldots, f_n{:}t_n \to t_n'$, it is possible to derive a *Struct* coercion $\Sigma \vdash t \lhd_s t' \rightsquigarrow c$, where neither t nor t' are among the types t_i, t_i' that are mentioned in Σ. Thus, the simple strategy of enumerating all possible coercion paths between types that are mentioned in Σ, which worked well for *Trans* (§4.2), cannot be applied to *Struct*. In the remainder of this section, we show how to approximate $\mathsf{NAC}_{\pi}(s, \Sigma)$ by imposing a slightly stronger constraint on Σ that is efficiently decidable.

Our approach is to reduce the problem of deciding $\mathsf{NAC}_{\pi}(s, \Sigma)$ to deciding a related property of the simpler *Trans* relation. We begin by introducing some convenient notation to describe the structure of a coercion derivation. We write $\Sigma \vdash t \lhd_r t_1 \lhd_s t' \rightsquigarrow c$ when we can decompose a derivation $\Sigma \vdash t \lhd_s t' \rightsquigarrow c$ into a segment $\Sigma \vdash t \lhd_r t_1 \rightsquigarrow c_r$ that uses transitive rule (CS-PrimTrans), and a segment $\Sigma \vdash t_1 \lhd_s t' \rightsquigarrow c_s$ that uses one of the structural rules (CS-FunTrans) or (CS-PairTrans). We elide the coercion term c when it is unimportant. This notation extends naturally to composition of longer sequences of coercions; e.g., we sometimes write $\Sigma \vdash t_1 \lhd_r t_2 \lhd_s t_3 \lhd_r t_4$ etc. Finally, for compactness, we write the application of a binary type constructor $T\ t_1\ t_2$ to stand for either the type $t_1 \times t_2$ or $t_1 \to t_2$.

$$\begin{array}{rccccccccccc} 1. & t & \lhd_r & \ldots & \lhd_r & (T\, t_1\, t_2) & \lhd_s & (T\, t_1'\, t_2') & \lhd_r & \ldots & \lhd_r & t' \\ \neg 2. & t & \lhd_r & \ldots & \lhd_r & \ldots & \lhd_r & \ldots & \lhd_r & \ldots & \lhd_r & t' \\ \neg 3. & t & \lhd_r & \ldots & \lhd_r & (S\, s_1\, s_2) & \lhd_s & (S\, s_1'\, s_2') & \lhd_r & \ldots & \lhd_r & t' \end{array}$$

$$\begin{array}{l} \mathsf{NAC}^*_{\pi}(s, \Sigma) \iff \\ \forall T, t_1, t_2, t_1', t_2', t, t', S, s_1, s_2, s_1', s_2'. \\ \quad \Sigma \vdash t \lhd_r (T\ t_1\ t_2) \lhd_s (T\ t_1'\ t_2') \lhd_r t' \wedge \\ \quad \Sigma \not\vdash (T\ t_1\ t_2) \lhd_r (T\ t_1'\ t_2') \wedge \\ \quad \{T, t_1, t_2, t_1', t_2'\} \neq \{S, s_1, s_2, s_1', s_2'\} \Rightarrow \\ \qquad \Sigma \not\vdash t \lhd_r t' \ \wedge\ \Sigma \not\vdash t \lhd_r (S\ s_1\ s_2) \lhd_s (S\ s_1'\ s_2') \lhd_r t' \end{array}$$

Figure 6. $\mathsf{NAC}^*_{\pi}(s, \Sigma)$: non-redundant subtyping paths

Our reduction of $\mathsf{NAC}_{\pi}(s, \Sigma)$ relies on the assumption that $\mathsf{NAC}_{\times}(r, \Sigma)$ holds. Under the constraint $\mathsf{NAC}_{\times}(r, \Sigma)$, the problem of deciding the uniqueness of arbitrary subtyping paths is considerably easier. If, for the moment, we focus only on product types, we can decide $\mathsf{NAC}_{\pi}(s, \Sigma)$ by only considering coercion derivations of the form $\Sigma \vdash t \lhd_r t_1 \times t_2 \lhd_s t_1' \times t_2' \lhd_r t'$. That is, under $\mathsf{NAC}_{\times}(r, \Sigma)$, the only viable coercion generation derivations that use a structural rule begin with a (possibly empty) prefix that coerces t to $(t_1 \times t_2)$ using (CS-PrimTrans); then, a single application of a structural rule coerces $(t_1 \times t_2)$ to $(t_1' \times t_2')$; and, finally, a (possibly empty) transitive suffix coerces the latter type to the goal t'. A coercion derivation that is not of this form, such as $\Sigma \vdash t \lhd_r (t_1 \times t_2) \lhd_s (t_1' \times t_2') \lhd_r (s_1 \times s_2) \lhd_s (s_1' \times s_2')$, violates $\mathsf{NAC}_{\times}(r, \Sigma)$, since it includes a primitive coercion between distinct pair types, i.e., $\Sigma \vdash t_1' \times t_2' \lhd_r s_1 \times s_2$.

Using this insight, we formulate a necessary condition for strong non-ambiguity, $\mathsf{NAC}^*_{\pi}(s, \Sigma)$, in Figure 6. The grey box at the top of the figure illustrates the condition on Σ that we aim to decide. On the first line, we show a derivation from t to t' that makes use of the *Struct* judgment in the form $(T\ t_1\ t_2) \lhd_s (T\ t_1'\ t_2')$. Whenever Σ admits such a derivation, for non-ambiguity, we require that the use of the structural rule in the first derivation is essential to coercing t to t'. We want to rule out derivations, such as the one shown on line 2, which can coerce t to t' using primitive coercions only. Additionally, we require that derivations of the form shown on line 3 are also not admitted by Σ, i.e., there must be at most one way to use structural rules to coerce t to t'.

$$\begin{array}{rlll}\Sigma & = & b{:}Bool \to Dyn, & \bar{b}{:}Dyn \to Bool,\\ & & i{:}Int \to Dyn, & \bar{i}{:}Dyn \to Int,\\ & & f{:}(Dyn \to Dyn) \to Dyn, & \bar{f}{:}Dyn \to Dyn \to Dyn\\ & & p{:}(Dyn \times Dyn) \to Dyn, & \bar{p}{:}Dyn \to Dyn \times Dyn\end{array}$$

$$\begin{array}{ll}\Sigma; \cdot \vdash_{r,s} (\lambda x{:}Dyn.x)\ 1 & \rightsquigarrow (\lambda x{:}Dyn.x)\ (i\ 1) : Dyn\\ \Sigma; \cdot \vdash_{r,s} 1\ \mathsf{true} & \rightsquigarrow ((\bar{f} \circ i)\ 1)\ (b\ \mathsf{true}) : Dyn\end{array}$$

Figure 7. Dynamic typing: coercion set and example derivations

Deciding $\mathsf{NAC}^*_\pi(s, \Sigma)$ is relatively straightforward. As in §4.2, we construct a coercion graph G_Σ, and for each pair of types t_1 and t_2 in the graph, we identify the sets of types $R_1 = \{(T\ t\ t') \mid \mathsf{Reachable}(G_\Sigma, t_1, (T\ t\ t'))\}$ and $R_2 = \{(S\ s\ s') \mid \mathsf{Reachable}(G_\Sigma, (S\ s\ s'), t_2)\}$. Next, for each pair of types t, t' in the cartesian product $R_1 \times R_2$, we check that at most one pair is related by the *Struct* judgment. That is, we construct the set $R = \{c \mid \Sigma \vdash t \lhd_s t' \rightsquigarrow c\ \wedge\ t, t' \in R_1 \times R_2\}$ and check that $|R| \leq 1$. If R contains more than one element, then Σ must violate $\mathsf{NAC}^*_\pi(s, \Sigma)$. If R is a singleton, and if t' is reachable from t in G_Σ, then once again, we have detected a violation of $\mathsf{NAC}^*_\pi(s, \Sigma)$.

In a system that includes only product types, it is possible to show that the conditions $\mathsf{NAC}^*_\pi(s, \Sigma)$, $\mathsf{NAC}_\times(r, \Sigma)$, $\mathsf{NAC}_\pi(r, \Sigma)$ and $\mathsf{NAC}_{app}(r, s, \Sigma)$ are both necessary and sufficient for strong non-ambiguity. A similar property can be shown in a setting with only function types. However, when we include both function and product types in the same system (or, for that matter, structural coercions for arbitrary type constructors) then the four NAC-conditions above are necessary, but not sufficient, for strong non-ambiguity. The problem is that our reduction of coercion derivations to a form that contains a *Trans* prefix, a single application of a *Struct* coercion, followed by a *Trans* suffix is not valid when both function and product types are present. For example, the derivation $\Sigma \vdash t \lhd_r (t_1 \times t_2) \lhd_s (t_1' \times t_2') \lhd_r (s_1 \to s_2) \lhd_s (s_1' \to s_2')$ is admissible since the primitive coercion $\Sigma \vdash (t_1' \times t_2) \lhd_r (s_1 \to s_2)$ does not violate $\mathsf{NAC}_\times(r, \Sigma)$. To remedy this, we propose a strengthening of $\mathsf{NAC}_\times$ and NAC_{app} to $\mathsf{NAC}_T(r, \Sigma)$, a condition that requires that each type t be coercible to at most one type of the form $(T\ t_1\ t_2)$. The definition below makes this condition precise. The theorem that follows establishes that efficiently decidable conditions $\mathsf{NAC}^*_\pi(s, \Sigma)$ and $\mathsf{NAC}_T(r, \Sigma)$ are a conservative approximation of the necessary condition for non-ambiguity, $\mathsf{NAC}_\pi(s, \Sigma)$.

Definition 12 ($\mathsf{NAC}_T(d, \Sigma)$: coercions to a type constructor)**.**
$\forall t, T, t_1, t_2, S, s_1, s_2, c, c'.$
$$\begin{array}{l}\Sigma \vdash t \lhd_d (T\ t_1\ t_2) \rightsquigarrow c\ \wedge\ \Sigma \vdash t \lhd_d (S\ s_1\ s_2) \rightsquigarrow c'\ \Rightarrow\\ \quad T = S \wedge t_1 = s_1 \wedge t_2 = s_2 \wedge c = c'\end{array}$$

Theorem 13 (Sufficiency of $\mathsf{NAC}^*_\pi(s, \Sigma)$)**.**

$$\forall \Sigma, \Gamma. \mathit{NAC}_\pi(r, \Sigma) \wedge \mathit{NAC}_T(r, \Sigma) \wedge \mathit{NAC}^*_\pi(s, \Sigma) \Rightarrow \mathit{NAC}_\pi(s, \Sigma)$$

5.5 Application: Dynamic and Gradual typing

A typical implementation of the untyped lambda calculus (e.g., for Scheme) tags each class of run-time value, and the interpreter checks for the appropriate tag before the value can be used. For example, the interpreter reduces the application $(\lambda x.x)\ 1$ after it confirms the left term $\lambda x.x$ has a function tag, but fails to reduce $(1\ \mathsf{true})$ when it finds that 1 has a non-function tag.

Henglein [1994] observed that tagging and untagging operations can be made explicit in a typed source language, where a standard untyped term is automatically translated to a typed term prior to evaluating it. The translation can avoid some redundant tag checks, improving on a naïve interpreter that would always implicitly perform a check. We can implement this idea directly.

The typed language extends a standard typed lambda calculus with the type Dyn for classifying untyped terms, along with a pair of coercions for each type constructor tc. The first converts a term of type $tc(Dyn, ..., Dyn)$ to Dyn by tagging it with tc. The second converts Dyn back to $tc(Dyn, ..., Dyn)$ by removing the expected tag tc, but fails if the term has some other tag instead. The Σ shown in Figure 7 enumerates such coercions: f and $\bar{f}$ for the $\to$ constructor; p and $\bar{p}$ for the $\times$ constructor; b and $\bar{b}$ for the $Bool$ base type (a nullary constructor); and i and $\bar{i}$ for the Int base type.

Given a term in the source language $\hat{e} ::= x \mid \lambda x{:}Dyn.\hat{e} \mid (\hat{e}_1, \hat{e}_2) \mid \mathsf{proj}_i(\hat{e})$, the same as our typed language e, but without explicit casts or ascriptions and where all bound-variable type annotations are Dyn, we perform coercion insertion using Σ from Figure 7. Derivations for the above examples appear at the bottom of the figure (with the trailing ids in the composed coercions elided for perspicuity). It is easy to see that coercion insertion will always succeed, and thus all dynamically-typed terms will be accepted, since Σ permits any type to be coerced to any other type. It is also worth noting that the translation eliminates some unnecessary tag checks. For the first example, an interpreter would naïvely tag the lambda term and then immediately untag it at the application, whereas our translation leaves the term alone. We have not proved that our translation optimally eliminates tag checks (producing a *minimal completion* in the terminology of Henglein), but plan to explore this issue in future work.

It is fairly easy to prove that the Σ of Figure 7 satisfies $\mathsf{NAC}_\times(r, \Sigma)$, $\mathsf{NAC}_{app}(r, s, \Sigma)$, $\mathsf{NAC}_\pi(r, \Sigma)$, and $\mathsf{NAC}_\pi(s, \Sigma)$, and thus, by Theorem 8, Σ admits only strongly unambiguous rewritings. However, our choice of Σ here also reveals the imprecision of our approximation of $\mathsf{NAC}_\pi(s, \Sigma)$ using $\mathsf{NAC}_T(r, \Sigma)$ and $\mathsf{NAC}^*_\pi(s, \Sigma)$. In particular, although we have $\mathsf{NAC}_\pi(s, \Sigma)$, because $\bar{f}$ and $\bar{p}$ are both in Σ, $\mathsf{NAC}_T(r, \Sigma)$ does not hold. While it is relatively straightforward to weaken $\mathsf{NAC}_T(r, \Sigma)$ to account specially for coercion sets that resemble the particular Σ here, a precise necessary and sufficient condition for non-ambiguity, which is still efficiently decidable for arbitrary Σ, remains an open question.

Gradual Typing. Siek and Taha [2006] introduced gradual typing as an approach to allowing programmers to mix dynamic and statically typed code. We can encode Siek and Taha's approach by viewing it as a generalization of the dynamic typing problem just considered. First, we allow source terms to be annotated with types other than Dyn; i.e., our source language becomes $\hat{e} ::= x \mid \lambda x{:}t.\hat{e} \mid (\hat{e_1}, \hat{e_2}) \mid \mathsf{proj}_i(\hat{e})$, where t is as in Figure 1 and base types include Dyn, Int, and $Bool$, as above. Second, we initialize the π parameter for coercion generation to prevent transitive compositions of coercions via Dyn. In particular, for *Trans* we change (CC-InitPath) as follows (and make a similar change to (CS-Init) for *Struct*):

$$\text{CC-InitPath'}\ \frac{\begin{array}{c}t' = Dyn \Rightarrow \pi = \{t\}\\ t' \neq Dyn \Rightarrow \{t, Dyn\}\\ \Sigma \vdash^\pi t \lhd_r t' \rightsquigarrow c\end{array}}{\Sigma \vdash t \lhd_r t' \rightsquigarrow c}$$

With this change, for *Struct* we have that for all t, $\Sigma \vdash Dyn \lhd_s t \rightsquigarrow c$ and $\Sigma \vdash t \lhd_s Dyn \rightsquigarrow c'$ for some c, c', but we do *not* have $\Sigma \vdash t \lhd_s t' \rightsquigarrow c''$ for all t', thanks to our initialization of π. As a result, *Struct* coercion generation matches Siek and Taha's *type consistency* relation, and thus ensures that terms not ascribed a Dyn type receive a useful degree of static checking, identifying some type errors and optimizing away some unnecessary coercions.

Consider once again our examples from Figure 7. For $(1\ \mathsf{true})$, the rewriting hinges on proving $\Sigma \vdash Int \lhd_r Dyn \to Dyn \rightsquigarrow c$, where $c \equiv id \circ \bar{f} \circ i$. But $\Sigma \not\vdash Int \lhd_r Dyn \to Dyn \rightsquigarrow c$ for our modified definition because Dyn in the initial π prevents

Open types	τ	$::=$	$\alpha \mid b \mid \tau_1 \rightarrow \tau_2 \mid \tau_1 \times \tau_2$
HM types	σ	$::=$	$\forall\vec{\alpha}.\tau$
HM Coercions	Σ	$::=$	$\cdot \mid \Sigma, f{:}\sigma \mid \Sigma, f{:}t$
Target language	c, m	$::=$	$\ldots \mid f[\vec{t}\,]$
Coercion path	π	$::=$	$\{c_1, \ldots, c_n, t_1, \ldots, t_k\}$
Substitutions	θ	$::=$	$\{(\alpha_1, t_1), \ldots, (\alpha_n, t_n)\}$
Index on relation	pq	$::=$	p (*PolyTrans*) $\mid q$ (*PolyStruct*)

$$\frac{FV(\tau_1) = \vec{\alpha}}{\vdash \forall\vec{\alpha}.\tau_1 \rightarrow \tau_2}$$

$$\text{CPQ-PrimTrans}\ \frac{\begin{array}{cc} f : \forall\vec{\alpha}.\tau_1 \rightarrow \tau_2 \in \Sigma & \tau_1 \approx t : \theta \\ t_2 = \theta(\tau_2) & \Sigma \vdash_{a'}^{\pi \uplus \{t_2, f\}} t_2 \lhd_{pq} t' \rightsquigarrow c \end{array}}{\Sigma \vdash_a^{\pi} t \lhd_{pq} t' \rightsquigarrow c \circ (f[\theta(\vec{\alpha})])}$$

Figure 8. Polymorphic coercions (*PolyTrans* and *PolyStruct*)

composing $\bar{f}$ and i. Thus we would reject this type-incorrect source term. The second example is the same with either definition. But if we annotate the bound variable we eliminate the need for the inserted coercion: $\Sigma; \cdot \vdash_{r,s} (\lambda x{:}Int.x)\ 1 \rightsquigarrow (\lambda x{:}Int.x)\ 1 : Int$.

6. Polymorphic coercions

This section presents the final enhancement to our coercion generation definition: the addition of primitive coercions with Hindley-Milner (HM)-style polymorphic types. The added expressiveness of HM types means we can program with object proxies—e.g., thunks and related constructs as described in the Introduction—as if they were the underlying objects themselves, and coercions will be inserted as necessary to mediate access. We illustrate the utility of HM coercions by encoding *provenance tracking*, such that a proxy is used to track the provenance of its underlying object. Provenance tracking is particularly important in queries to curated scientific databases [Cheney et al. 2007].

6.1 Coercion generation

We extend both the *Trans* and *Struct* coercion generation definitions so that primitive coercions in Σ can have HM polymorphic types $\forall\vec{\alpha}.\tau$, where open types τ are as types t but may contain (quantifiable) type variables α. We call the extended coercion generation definitions *PolyTrans* and *PolyStruct*. For both systems, coercion insertion produces target terms m that may contain instantiated coercion applications $f[\vec{t}\,]$. The source language e is the simply typed lambda calculus, as before (Figure 1). Polymorphically typed source language terms create significant complication, so we leave their consideration to future work.

The *PolyStruct* judgment is identical to *Struct*, except for one additional rule, (CPQ-PrimTrans) shown in Figure 8—all the rules in the resulting judgment are indexed by q. *PolyTrans* is defined likewise as an extension to *Trans*, using index p.[4] Rule (CPQ-PrimTrans) has three differences from (CS-PrimTrans). First, to use a polymorphic coercion with type $\forall\vec{\alpha}.\tau_1 \rightarrow \tau_1'$, the second premise requires that t, the source type of the coercion, is unifiable with τ_1 according to substitution θ. Next, the third premise applies θ to τ_2 to produce *closed* type t_2 which is then used in the fourth premise to, as before, transitively produce a coercion to the target type t', adding t_2 to the path π. That t_2 is closed implies $dom(\theta) \supseteq FV(\tau_2)$; to ensure this is always the case, polymorphic types σ in $rng(\Sigma)$ must satisfy the condition $\vdash \sigma$, which requires all bound type variables occur in the domain of a polymorphic coercion. The alternative of allowing $\theta(\tau_2)$ to contain free type variables adds significant complication to the rules, while the $\vdash \sigma$ restriction is satisfied by most applications we have considered. Finally, notice in the last premise we also add f to the path π; we explain why below.

PolyStruct has the same problem as *Struct* in that for some Σ it could be used to generate an infinite number of function or product type coercions from a given source type (Section 5.2). To avoid this problem, as before, we can limit the use of *PolyStruct* to the right-hand side of a function application, i.e., by only considering a coercion insertion judgment of the form $\Sigma \vdash_{p,q} e \rightsquigarrow m : t$ (or $\Sigma \vdash_{p,p} e \rightsquigarrow m : t$, but not $\Sigma \vdash_{q,q} e \rightsquigarrow m : t$).

Interestingly, notice that without adding f to π in the fourth premise of (CPQ-PrimTrans), it is also possible for *PolyTrans* to generate an infinite number of product and function coercions. For example, with $f : \forall\alpha.\alpha \rightarrow (\alpha \times \alpha)$, we can generate $\Sigma \vdash Int \lhd_p Int \times Int \rightsquigarrow f[Int]$, and $\Sigma \vdash Int \lhd_p (Int \times Int) \times (Int \times Int) \rightsquigarrow (f[Int \times Int]) \circ (f[Int])$, and so on. Adding f to π ensures that in a transitively generated coercion chain $c_1 \circ \ldots \circ f[\vec{t}\,] \circ \ldots \circ c_n$, no $c \in c_1 \ldots c_n$ will be $f[\vec{t'}]$. Note that f may still appear in structural coercions used to construct some c_i within the chain, since π is reset when generating coercions for the component types. Eliminating this restriction, should doing so become necessary, remains future work.

Example: Dynamic proxies. Polymorphic coercions are useful for implementing dynamic proxies, such as thunks (for lazy evaluation) and futures (for parallel evaluation) [Pratikakis et al. 2004]. Coercions for thunks can be defined as follows: $\Sigma = lazy{:}\forall\alpha.(Unit \rightarrow \alpha) \rightarrow Lazy\ \alpha, force{:}\forall\alpha.Lazy\ \alpha \rightarrow \alpha$. The *lazy* coercion injects a thunk $Unit \rightarrow \alpha$ into a $Lazy\ \alpha$, while *force* converts a $Lazy\ \alpha$ to a α by evaluating it. Thus we have

$$\begin{array}{l} \Sigma; \cdot \vdash (\lambda y{:}Lazy\ Int.y + 1)\ (\lambda x{:}Unit.e) \rightsquigarrow \\ \qquad (\lambda y{:}Lazy\ Int.(force[Int]\ y) + 1)\ (lazy\ \lambda x{:}Unit.e) : Int \end{array}$$

Notice that the programmer must indicate the expression e to evaluate lazily by "thunkifying" it manually. We could imagine instead using something like OCaml's `lazy` e annotation from which a syntax manipulation tool like camlp4 would perform the thunkification and insert the *lazy* coercion. Coercion insertion complements such an approach, since a tool like camlp4 cannot automatically insert calls to *force* since doing so precisely requires type information.

6.2 Strong non-ambiguity

Each of the necessary and sufficient strong non-ambiguity constraints presented in Section 3 also apply to rewritings that make use of polymorphic coercions. Developing an efficient algorithm for deciding these conditions is an open problem. Here we discuss some of the difficulties presented by this problem.

To decide $\mathsf{NAC}_\pi(p, \Sigma)$, we need to detect when two coercions in Σ overlap. Consider, for example, a set Σ that contains $f{:}\forall\alpha.\alpha \rightarrow (\alpha \times Int)$, $g{:}Int \rightarrow (Int \times Int)$, and $h{:}\forall\alpha.\alpha \rightarrow (Int \times \alpha)$. This set is ambiguous since the type of g is an instance of the type of f. However, simply removing g from Σ does not eliminate the ambiguity, since, although neither f nor h is an instance of the other, there are still two ways to coerce an *Int* to a pair $(Int \times Int)$. When the structural rules of *PolyStruct* are added, the problem is harder still.

Deciding $\mathsf{NAC}_\times(p, \Sigma)$ (and also $\mathsf{NAC}_{app}(p, q, \Sigma)$) is also challenging. For example, f, above, provides a way to coerce one product type to another, i.e., $\Sigma \vdash (Int \times Int) \lhd_p ((Int \times Int) \times Int) \rightsquigarrow f[Int \times Int]$. To detect a violation of $\mathsf{NAC}_\times$, one must find instantiations of coercions' type variables that enable coercion paths from a type t to distinct product types. While the offending instantiation

[4] Technically, the version of (CPQ-PrimTrans) for *PolyTrans* is written $\Sigma \vdash^{\pi} t \lhd_p t' \rightsquigarrow c$; the indices a and a' that subscript the turnstile in the rule definition in Figure 8 are dropped.

enzymes			
pid	*name*	*mw*	*prov*
1	flavodoxin	19.7	PLib1
2	ferrodoxin	12.3	PLib2
…	…	…	…

xrefs	
rid	*pid*
1	1
1	2
…	…

reactions		
rid	*name*	*prov*
1	G(1)	Lab1
2	R(8)	Lab2
…	…	…

Query: Sum of enzyme weights in each reaction			
Reaction name		Sum weights	
name	*name prov*	*weight*	*weight prov*
G(1)	Lab1	$19.7 + 12.3$	PLib1, PLib2
R(8)	Lab2	…	…
…	…	…	…

Figure 9. A query on a database of biochemical reactions

for α in our example is relatively easy to find, we have not developed a general decision procedure for this purpose.

Despite the lack of an algorithm for deciding $\mathsf{SNA}(p, q, \Sigma)$, with only a little guidance from the programmer even ambiguous sets of polymorphic coercions can be put to good use. The next example illustrates how a form of dynamic information flow tracking can be encoded as an instance of polymorphic coercion insertion. Through the careful placement of a few type ascriptions, a programmer can direct the coercion insertion process to ensure that the program is always unambiguously rewritten.

6.3 Application: Provenance tracking

Cheney et al. [2007] have developed a semantics for tracking provenance in database queries, an application that is useful for both security and reliability. In this section, we borrow one of their examples and show how we can automatically rewrite queries with coercions that do the provenance tracking.

Figure 9 illustrates a database of biochemical records over which we wish to run queries and Figure 10 shows how we model this scenario. The database contains three tables: *enzymes*, *reactions* and *xrefs*. We represent each table as a list of tuples, and the typing environment at the top of Figure 10 binds variables for each of the tables.[5]

The *enzymes* table has three main columns: a primary key *pid*, the *name* of the protein in the enzyme, and the molecular weight *mw* of that protein. A final column, *prov*, records the provenance of the information in each row, e.g., the first row in the *enzymes* table was obtained from the protein library "PLib1". Our encoding of this table in Figure 10 gives *enzymes* the type *List* (*Prov protein*). The type abbreviation *protein* expands to a tuple with three fields, matching the first three columns of the table. The whole row is given type *Prov protein*—our encoding represents a t-typed value that is tagged with provenance metadata using the type *Prov t*. We leave the representation of the provenance metadata unspecified; here we think of it as a *String*.

The second table records a collection of biochemical *reactions* with a primary key *rid* and the *name* of the reaction. As with the *enzymes* table, each row in *reactions* is tagged with its provenance, say, the name of the laboratory that provided this information. In Figure 10, we represent this table using a value of type *List Prov*(*Int* ×*String*).

A final table, *xrefs*, cross-references the *reactions* and *enzymes* tables, indicating all the proteins involved in a particular reaction.

Our goal is to run a query that, for each reaction, computes a result that pairs the name of the reaction with the sum of the molecular weights of all the proteins that are used in that reaction. Of course, this query must also track provenance. So, as shown in Figure 9, the query result includes a row that pairs the reaction name "G(1)" with the name of the laboratory "Lab1"; and the sum $19.7 + 12.3$ paired with "Plib1, PLib2", to indicate that the result involved data from both these sources.

Our coercion insertion approach allows a programmer to focus solely on the core query semantics, without worrying about inserting coercions to track provenance. Programmers need only specify simple general purpose combinators for provenance tracking and our system can automatically retro-fit a query with provenance-tracking semantics. For example, a value of type *Prov t* can be used by the programmer at type t, and our system will insert the appropriate coercion. We first describe the structure of the source query (at the bottom-left of Figure 10) and then discuss the specific coercions that we use to rewrite the program. Our example uses the convenience of **let**-bindings, but we are careful to annotate all λ-bindings with their types. We also make use of **if**-expressions. A translation from this extended syntax to our core calculus is straightforward.

We write our *query* as a function over a single row in the reaction table and then, at line 13, apply this function to the *reactions* table. In the body of the *query*, we project out the *rid* and *name* fields from the reaction and then construct an *aggregate* query which is supposed to perform a summation of all the molecular weights of proteins by consulting the *xrefs* and *enzymes* tables. The result of the *query* is the name of the *reaction* paired with the aggregate query applied to the *xrefs* table.

The aggregate query is written in the style of a function that can be used to fold over a list. Its first argument *sum* is an accumulator; the second argument is a row from the *xrefs* table. In the body of the aggregate query, we first check if the reaction id in the *xref* record matches the reaction being processed by the query, i.e., this is the equivalent of checking a join constraint. If the check succeeds, we look up the related protein, project out its weight and add it to the accumulator. Notice that the typing environment provides a function *lookup* that retrieves a row in the *enzymes* table based on its primary key and a *plus* function to add floating point numbers.

We rewrite this query using a Σ that includes primitive polymorphic coercions to manipulate the *Prov t* and *List t* types. Notice that all of these coercions satisfy our well-formedness restriction on HM types. The *lift* function injects an arbitrary type α into the $Prov\ \alpha$ type (by pairing it with some default provenance information). The *lower* coercion allows nested provenance wrappers to be collapsed. The coercion *asfun* allows a function f that has been lifted into the $Prov\ (\alpha \rightarrow \beta)$ type to be applied to a provenance-tracked value x of type $Prov\ \alpha$. Since the value returned by a function depends both on the provenance of the function and the provenance of the argument, *asfun* is expected to tag the resulting β-value with the provenance of both f and x. (It is interesting to note that the *lift* and *asfun* functions together make our *Prov t* type an idiom [Lindley et al. 2008].) The *asprod* coerces a provenance-tracked product into a product of provenance-tracked values. Finally, the coercion set includes functions *rmap* and *foldl*, both standard combinators on lists. Notice, however, that the order of arguments in these functions is important; we cannot swap the first two arguments in *rmap*, for example, since $\not\vdash \forall\alpha, \beta. List\ \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow List\ \beta$.

The coercion set Σ that we use for this example does not satisfy the necessary conditions for non-ambiguity. For example, a product type $t_1 \times t_2$ can be coerced both to itself and, via *asprod* ∘ *lift*, to $Prov\ t_1 \times Prov\ t_2$.Nevertheless, with a single type ascription for the entire query (shown on line 13 of the source program), the programmer can pick one of the several possible rewritings to resolve the ambiguity. For brevity, we leave out the type instantiations on polymorphic coercions.

[5] For this example, we assume that base types also include type constructor applications like *List t* and *Prov t*. We do not assume the presence of any structural rules that allow coercions to be lifted into these types.

Typing environment Γ: Tables declared as lists of records rather than products for clarity; a type abbreviation, and some utility functions.

$enzymes{:}List\ (Prov\ \text{protein})$	$\text{protein} \equiv (\text{pid}{:}Int \times (\text{name}{:}String \times \text{mw}{:}Float))$
$reactions{:}List\ (Prov\ (\text{rid}{:}Int \times \text{name}{:}String))$	$lookup{:}Int \to List\ (Prov\ \text{protein}) \to Prov\ \text{protein}$
$xrefs{:}List\ (\text{rid}{:}Int \times \text{pid}{:}Int)$	$plus{:}Float \to Float \to Float, neq{:}Int \to Int \to Bool$

Coercion set Σ: $Prov\ t$ is the type of a t-value tagged with provenance metadata; we omit the data constructors for $List\ t$ and $Prov\ t$.

$lift{:}\forall\alpha.\alpha \to Prov\ \alpha$	$lower{:}\forall\alpha.Prov\ (Prov\ \alpha) \to Prov\ \alpha$
$asfun{:}\forall\alpha,\beta.Prov\ (\alpha \to \beta) \to Prov\ \alpha \to Prov\ \beta$	$asprod{:}\forall\alpha,\beta.Prov\ (\alpha \times \beta) \to Prov\ \alpha \times Prov\ \beta$
$rmap{:}\forall\alpha,\beta.(\alpha \to \beta) \to List\ \alpha \to List\ \beta$	$foldl{:}\forall\alpha,\beta.(\alpha \to \beta \to \alpha) \to \alpha \to List\ \beta \to \alpha$

Source program

```
 1 let query =
 2   λreaction:(Int ×String).
 3     let rid = proj_1 reaction in
 4     let name = proj_2 reaction in
 5     let aggregate=
 6       λsum:Prov Float. λxref:(Int ×Int).
 7         if neq (proj_1 xref) rid then sum else
 8         let pid = proj_1 xref in
 9         let protein = lookup pid enzymes in
10         let wgt = proj_2 (proj_1 protein) in
11           plus wgt sum in
12       (name, aggregate 0 xrefs) in
13 query reactions : List Prov (Prov String ×Prov Float)
```

Target (rewritten) program

```
 1 let query (* (Int ×String) → (Prov String ×Prov Float) *) =
 2   λreaction:(Int ×String).
 3     let rid = proj_1 reaction in
 4     let name = proj_2 reaction in
 5     let aggregate (* Prov Float → (Int ×Int) → Prov Float *) =
 6       λsum:Prov Float. λxref:(Int ×Int).
 7         if neq (proj_1 xref) rid then sum else
 8         let pid = proj_2 xref in
 9         let protein = lookup pid enzymes in
10         let wgt = proj_2 (asprod (proj_1 (asprod protein))) in
11           (asfun ((asfun ∘ lift) plus) wgt) sum in
12       (lift name, (foldl aggregate) (lift 0) xrefs) in
13 ((rmap ∘ asfun∘ lift) query) reactions : List Prov (Prov String ×Prov Float)
```

Figure 10. Equipping database queries with provenance tracking

We describe the rewritten query, given in the lower right of the figure, from inside out, starting with the body of the *aggregate* query. The first rewrite occurs at line 10. To project the *mw* field from *protein*, we first coerce the $Prov$ protein to a product *Prov Int* ×(*Prov*(*String* ×*Float*)) using *asprod*, then coerce it again before projecting out the weight at the type *Prov Float*. To add two *Prov Float* values, at line 11 we need to lift the *plus* function into the *Prov* idiom using *lift* and then *asfun*. Since *plus* is curried, the partial application (*asfun* ∘ *lift plus*) *wgt* is given the type *Prov* (*Float* → *Float*). With another application of the *asfun* we may apply *sum* to get a value of type *Prov Float*. The type of the rewritten *aggregate* query is shown as a comment on line 5.

To apply the *aggregate* query to the *xrefs* table, we coerce it using *foldl* to *Prov Float* → *List* (*Int* ×*Int*) → *Prov Float*. We apply the result to the constant 0 lifted into the *Prov Float* type, and again to the *xrefs* table itself. By lifting *name* into the *Prov* type, we can give *query* the type shown as a comment on line 1. Applying *query* to *reactions* is similar to the application of *plus* at line 11. The coercion *rmap* ∘ *asfun* ∘ *lift* coerces *query* to the type *List* (*Prov* (*Int* ×*String*)) → *List* (*Prov* (*Prov String* ×*Prov Float*)) which is the type necessary to apply it to run the query over *reactions* table.

It is worth considering the precision that our method provides when tracking provenance. Swamy et al. [2008] have shown that this encoding of provenance is sufficient to establish *dependency correctness*, a property due to Cheney et al. [2007]. This is an extremely strong property and can be used to track both explicit and implicit dependences, in a manner related to noninterference-based information flow [Sabelfeld and Myers 2003]. However, if not used carefully, our encoding can produce provenance results that claim, for example, that an element of a result set depends on all other data in the database. Suppose, for example, that instead of using *lookup* primitive on the *protein* table, we performed a scan of the table (using *rmap*) to explicitly search for the entry with a primary key *pid*. According to dependency correctness, every aggregate molecular weight in the result would depend (negatively) on every row in the *enzymes* table. While this is technically correct, it is not very useful. Handling negative and implicit dependences is a challenge even faced by custom provenance tracking system, e.g., in the system of Cheney et al. [2007]. Our approach allows programmers to provide special functions like *lookup* to control how aggressively provenance information is tracked through a query.

7. Related work

Compared to the prior work on coercive subtyping our work includes, first, a more careful treatment of ambiguity and, second, provides greater expressiveness. With regards to the first, the structure of our judgments allows us to treat ambiguity purely syntactically, which makes it indifferent to the semantics of coercions, e.g., even if they are effectful. All prior work we know of either allows ambiguous rewritings but proves they are confluent (typically assuming coercions, or the source language, is not effectful), or preferentially selects a particular rewriting in an ad hoc way (e.g., a program that type checks without any coercions is left untouched), which is less intuitive [Henglein 1994, Luo 1996, 1999, Luo and Luo 2005, Luo 2008]. Our conditions for strong nonambiguity are necessary and sufficient; prior works find sufficient conditions (e.g., Saïbi [1997], and Luo and Kießling [2004]), but these are more restrictive than necessary, ruling out some useful, unambiguous rewritings.

Our coercions are more expressive in that they include transitivity, structural rules, and polymorphism. We have not found this combination, with the same degree of flexibility, in the literature. For example, the coercion calculus of [Henglein 1994] includes structurally and transitively-composed coercions, and work by Luo and Luo [2005], building on the works of Aczel [1995] and Barthe [1996], does likewise. More recently, Luo [2008] has considered HM polymorphism, but without transitivity or structural rules. Saïbi's coercion system for Coq does support transitivity, structural rules, and polymorphism, but he imposes a "uniformity" restriction that severely limits the use of polymorphism; e.g., coercions of the form $\forall\alpha,\beta.Prov(\alpha \to \beta) \to Prov\ \alpha \to Prov\ \beta$ cannot be expressed in his system.

Another difference in expressiveness when comparing our approach to coercive subtyping is that we wish to handle applications where coercions may have operational effect—this is partly the reason why it is particularly important to ensure that coercion insertion is unambiguous. Traditional coercive subtyping treats subtyping as an identity coercion [Sulzmann et al. 2007, Breazu-Tannen et al. 1991], or expects coercions to be total functions [Saïbi 1997]. This makes prior works unsuitable for applications such as provenance and gradual typing where the coercions necessarily have computation effect (to store provenance tracking information in the database, or for downcasts to fail). Henglein [1994] considers dynamic typing as a coercion insertion problem (Section 5.5) where inserted coercions perform dynamic checks that may fail. He develops a rewriting system that aims to produce a canonical, optimally safe rewriting, assuming a particular semantics for coercions. Thus his work is less general than ours, but provides a stronger guarantee for this application.

We consider a simply-typed source language while other works consider source terms with dependent types [Saïbi 1997], Fsub source terms [Breazu-Tannen et al. 1991], or ML [Luo and Kießling 2004]. Simply-typed source terms are useful in and of themselves, for example to apply this to rewriting database queries, or to retrofit monomorphic C code with security checks [Ganapathy et al. 2006]. Nevertheless, extending our work to handle a polymorphic source language is significant future work.

8. Conclusions and future work

In this paper we have defined a general framework for type-directed coercion insertion. One of the innovations of our framework is to directly address the problem of ambiguity and separately define mechanisms to curb ambiguity. We developed various versions of coercion generation that differ in the expressivity they provide to the rewriting system. To justify our work we have shown how a number of purpose-built rewriting schemes ranging from DSU to type systems to provenance tracking can be seen as specific instances of our general-purpose system.

In future work we should like to further explore the applicability of our work to type-directed rewriting problems considered in the literature, such as other forms of gradual typing [Wadler and Findler 2009, Siek et al. 2009] and hybrid typing [Flanagan 2006]. We should also like to consider our framework in the richer setting of dependent types. This is not just out of theoretical curiosity; our original motivation for this work, Fable [Swamy et al. 2008], is a dependently typed language for enforcing user-defined security policies. Indeed, the encoding of provenance tracking in §6.3 is closely related to an encoding that is used in Fable, although Fable's type system enables the provenance of a value to be made evident in its type, not just in its runtime representation.

Acknowledgments

The authors would like to thank Jeff Foster, Phil Wadler, Dimitrios Vytiniotis, Iulian Neamtiu, Polyvios Pratikakis, Peter Sewell, and the anonymous reviewers for helpful comments that led to significant improvements in the paper's presentation. Hicks was supported in part by NSF grants CCF-0541036, CCF-0524036, and CNS-0346989 and by Microsoft Research, Cambridge, while on sabbatical during the Fall of 2009, when much of this work was carried out.

References

P. Aczel. A notion of class for type theory, 1995. Unpublished manuscript.

G. Barthe. Implicit coercions in type theories. In *Proc. of Types workshop*, 1996.

V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. Inheritance as implicit coercion. *Information and Computation*, 93:172–221, 1991.

J. Cheney, A. Ahmed, and U. A. Acar. Provenance as dependency analysis. In *Proc. of DBPL*, 2007.

C. Flanagan. Hybrid type checking. In *Proc. of POPL*, 2006.

V. Ganapathy, T. Jaeger, and S. Jha. Retrofitting legacy code for authorization policy enforcement. *Proc. of Security and Privacy*, 2006.

F. Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22:197–230, 1994.

S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. In *Proc. of MSFP*, 2008.

Z. Luo. Coercions in a polymorphic type system. *Mathematical Structures in Computer Science*, 18(4):729–751, 2008.

Z. Luo. Coercive subtyping in type theory. In *Proc. of CSL*, 1996.

Z. Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, 1999.

Z. Luo and R. Kießling. Coercions in Hindley-Milner systems. In *Proc. of Types*, 2004.

Z. Luo and Y. Luo. Transitivity in coercive subtyping. *Information and Computation*, 197(1-2):122–144, 2005.

P. Pratikakis, J. Spacco, and M. Hicks. Transparent proxies for Java futures. In *Proc. of OOPSLA*, 2004.

A. Sabelfeld and A. C. Myers. Language-based information-flow security. *JSAC*, 21(1):5–19, 2003.

A. Saïbi. Typing algorithm in type theory with inheritance. In *Proc. of POPL*, 1997.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Proc. of Scheme and Functional Programming Workshop*, 2006.

J. G. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In *Proc. of ESOP*, 2009.

G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. *ACM TOPLAS*, 29(4), 2007.

M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *Proc. of TLDI*, 2007.

N. Swamy, B. J. Corcoran, and M. Hicks. Fable: A language for enforcing user-defined security policies. In *Proc. of Security and Privacy*, 2008.

N. Swamy, M. Hicks, and G. Bierman. A theory of typed coercions and its applications. Technical Report MSR-TR-2009-69, Microsoft Research, 2009.

P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *Proc. of ESOP*, 2009.