Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 196 (2008) 53–68
www.elsevier.com/locate/entcs

Coercive Subtyping via Mappings of Reduction Behaviour

Paul Callaghan[1,2]

Department of Computer Science
University of Durham
Durham, UK.

Abstract

This paper reports preliminary work on a novel approach to Coercive Subtyping that is based on relationships between reduction behaviour of source and target types in coerced terms. Coercive Subtyping is a superset of record-based subtyping, allowing so-called coercion functions to carry the subtyping. This allows many novel and powerful forms of subtyping and abbreviation, with applications including interfaces to theorem provers and programming with dependent type systems. However, the use of coercion functions introduces non-trivial overheads, and requires difficult proof of properties such as coherence in order to guarantee sensible results. These points restrict the practicality of coercive subtyping.

We begin from the idea that coercing a value v from type U to a type T intuitively means that we wish to compute with v as if it was a value in T, not that v must be converted into a value in T. Instead, we explore how to compute on U in terms of computation on T, and develop a framework for mapping computations on some T to computations on some U via a simple extension of the elimination rule of T. By exposing how computations on different types are related, we gain insight on and make progress with several aspects of coercive subtyping, including (a) distinguishing classes of coercion and finding reasons to deprecate use of some classes; (b) alternative techniques for proving key properties of coercions; (c) greater efficiency from implementations of coercions.

Keywords: Coercive subtyping, type theory, inductive families, implementation.

1 Introduction

Coercive subtyping is an abbreviation mechanism which handles mismatches of type between a value and its context of use with implicit insertion of functions to bridge the gap. It has many natural uses, such as providing a form of record subtyping on nested algebraic structures or conversions between simple types. In the context of dependent types and inductive families, coercions can be used for many other interesting purposes, not least modelling of semantics of words in natural languages

[1] This work is supported by the TYPES Working Group, a coordination action in the EU 6'th Framework programme.
[2] Email: mailto:p.c.callaghan@durham.ac.uk

doi:10.1016/j.entcs.2007.09.017

and providing a bridge between simply and dependently typed data. As a running example, consider the coercions between conventional lists and sized vectors. A vector can be used wherever a list is expected by implicitly coercing the vector with function $v2l$, or in the reverse direction with $l2v$ (though note the dependency to find the size of the vector):

$$v2l \ : \ (A : \mathit{Type})(n : \mathbb{N}) \ \mathit{Vec}\ A\ n \rightarrow \mathit{List}\ A$$

$$l2v \ : \ (A : \mathit{Type}) \qquad (l : \mathit{List}\ A)\ \mathit{Vec}\ A\ (\mathit{length}\ A\ l)$$

However, the use of coercions relies on non-trivial proofs of key properties such as coherence (uniqueness of result up to conversion) and transitivity elimination. Progress has been made on some sub-classes of coercions (between restricted groups of types), but establishing results *beyond* these classes and establishing them in a satisfactory (i.e., elegant and/or natural) way is an open problem. For practical purposes, such as flexible implementation within proof tools, such proofs should ideally be automatable and not require the user to provide difficult justifications. There are also issues of efficiency: the standard formulation implies some overheads during computation, e.g. conversion of at least some of a vector to a list before any computation can proceed. A final concern is with how intermediate computations on coerced values appear to the user of proof tools.

One view is that the standard formulation of coercive subtyping is too strong, in the sense that it allows very powerful coercion functions to be conceived but which are problematic for practical use. This view is based on the author's experience of earlier implementation and experimentation with many kinds of coercion function. The author also prefers that use of this intuitively 'natural' mechanism of coercion should not be limited by the need to consider or to understand non-trivial details of how classes of coercion interact. Too much complexity and the claim of being a flexible abbreviation mechanism is significantly weakened.

This paper explores one novel alternative. We start from the idea that coercion means that we wish to use a value v in source type U as if it was a value in target type T, or more specifically, that we wish to compute with v in terms of T. This intention does not force us to convert v into a representation in T. Instead, a mapping is constructed between the computation behaviours of U and of T in terms of the elimination operators of both types which allows easy transformation of operations on T into operations on U. This clearly avoids converting v to T, but we suggest there are several other important benefits, including (a) the links between source and target type are made explicit and are in terms which are fundamental to both types, so it is thus clearer why and how U can be treated as T; (b) making such information more explicit has several benefits in establishing key properties, not least simplifying some of the proofs; (c) it provides a framework to discuss and characterise different proposed coercions, and possibly to impose meaningful structure on sets of coercions.

The following summarises the key ideas and contributions. We first observe that summation on lists (of $\mathbb{N}$) can be translated to summation on vectors by modification of the arguments of the elimination operator. In fact, all functions on lists can be

converted to functions on vectors in exactly the same way. The modification reflects how the types are related and this is independent of actual functions. We aim to use this relationship to allow computation on a source value in terms of the target type, without converting the source value. A particular form of elimination operator is proposed to encode this relationship in a clear way: operators $\mathbf{E}_{XY}$ should take the parameter, motive, and branch function arguments of the eliminator for type X, but take a value in type Y as the elimination target, and should be defined in terms of $\mathbf{E}_Y$. For the vector-to-list example, this means:

$$\frac{C : \mathit{List}\ A \to \mathit{Type} \quad f_0 : C\ (\mathit{nil}_A) \quad f_1 : (a : A)(l : \mathit{List}\ A)C\ l \to C\ (\mathit{cons}_A\ a\ l)}{\mathbf{E}_{LV}\ A\ C\ f_0\ f_1 : (n : \mathbb{N})(v : \mathit{Vec}\ A\ n)C\ (\mathit{co}_V\ A\ n\ v)}$$

$$\mathbf{E}_{LV}\ A\ C\ f_0\ f_1 =_{\mathrm{df}} \mathbf{E}_V\ A\ ([n : \mathbb{N}][v : \mathit{Vec}\ A\ n]C\ (\mathit{co}_V\ A\ n\ v))\ f_0\ ([m : \mathbb{N}][x : A][v : \mathit{Vec}\ A\ m]f_1\ x\ (\mathit{co}_V\ A\ m\ v))$$

Which other combinations of types support the definition of a $\mathbf{E}_{XY}$ term? A conservative answer is the combinations whose conversion (from Y to X) is invertible. This covers many useful conversions, including functorial maps and natural conversions between different container types.

To use these terms, we propose that coercion becomes a two-stage process. Initially, a coerced term is marked (with a form of constructor), and when this reaches an eliminator of the target type, the $\mathbf{E}_{XY}$ term is used to map the computation to the source type. Hence no conversion (of value) takes place, and nothing happens until a computation acts on the coerced value. This process is clearly type-safe, although formal metatheory relating to canonicity and convertibility remain to be studied as further work. (Some preliminary remarks are made.) We briefly consider how the $\mathbf{E}_{XY}$ terms affect the proof of relevant properties. The key point is that composition via $\mathbf{E}_{XY}$ terms eventually reduces to elimination on the source type.

We first review the background type theory and some relevant earlier results on Coercive Subtyping. The central idea is presented through an expansion of the list and vector examples, then formalised as a relationship between elimination operators. Consequences of this formalisation are then explored, through further examples. The paper ends with comments on metatheory and implementation.

2 Preliminaries

2.1 Inductive families and their elimination

This paper concerns relationships between inductive families which are not tied to specific type theories, hence we assume a 'vanilla' dependent type theory. Briefly, there is a dependent product type $(x : K)K'$ (where x may occur free in K'), and we often write $K \to K'$ for dependent products with no dependency. *Type* is the sort of all types, i.e. $A : \mathit{Type}$ means A is a type. Notation $[x : K]k$ denotes λ-terms. The system includes an η rule, i.e. $[x : K]f\ x = f : (x : K)K',\ x \notin FV(f)$.

Inductive types [7,8,11] may be introduced through a schema [12], summarised as the grammar $\Theta = X \mid (x : K)\Theta \mid \Phi \to \Theta$ and $\Phi = (\overline{x} : \overline{K})X$. It identifies a class

of inductive types which recurse through strictly positive operators and specifies how the elimination operators and computation rules are formed for each type. X is a placeholder for the type being defined. Small types K, K_i are those which don't contain *Type*. Inductive type T is constructed from a sequence $\overline{\Theta}$ which represents the types of T's constructors. The types of the constructors and the elimination operator, and the computation rules for the constructors, are constructed by analysis of the schemata. Inductive types can also be *parametrized*, e.g. to give polymorphic lists. The type of the elimination operator and associated computation rules are as follows. (We adopt elements of natural deduction style from [17].)

$$\frac{C : \mathit{List}\ A \to \mathit{Type} \quad f_0 : C\ (\mathit{nil}\ A) \qquad f_1 : (x : A)(xs : \mathit{List}\ A)C\ xs \to C\ (\mathit{cons}\ A\ x\ xs)}{\mathbf{E}_L\ A\ C\ f_0\ f_1 : (z : \mathit{List}\ A)C\ z}$$

$$\mathbf{E}_L\ A\ C\ f_0\ f_1\ (\mathit{nil}\ A) = f_0$$

$$\mathbf{E}_L\ A\ C\ f_0\ f_1\ (\mathit{cons}\ A\ x\ xs) = f_1\ x\ xs\ (\mathbf{E}_L\ A\ C\ f_0\ f_1\ xs)$$

The result type of the elimination is determined by the *motive* argument (C in these examples) [17]. Computation over the constructors are handled by the "case functions" or "branch functions" (f_0 and f_1 above), one for each constructor. Finally, we have the 'target' z to eliminate, and the corresponding result $C\ z$.

Inductive families [8] are a generalisation of inductive types, where a *family* of types is inductively defined. An extended schema [12] replaces constant X with an indexed form $X\ \overline{q}$ and modifies the construction of operators to insert indices at appropriate places. These indices are different in nature from the parameters above: parameters are fixed for any instance of a type, but the indices may vary inside the value depending on how it has been constructed.

A standard example is the family of vectors, $\mathit{Vec} : \mathbb{N} \to \mathit{Type}$, indexed by length, with constructors $\mathit{vnil} : (A : \mathit{Type})\,\mathit{Vec}\ A\ \mathit{zero}$ and $\mathit{vcons} : (A : \mathit{Type})(n : \mathbb{N})A \to \mathit{Vec}\ A\ n \to \mathit{Vec}\ A\ (\mathit{succ}\ n)$. The cons operation only extends a vector by one unit of size. There are no other ways to build vectors. These constraints are reflected in the type and behaviour of the elimination operator.

$$\frac{A : \mathit{Type} \quad C : (n : \mathbb{N})\,\mathit{Vec}\ A\ n \to \mathit{Type} \quad f_0 : C\ \mathit{zero}\ (\mathit{vnil}\ A) \qquad f_1 : (n : \mathbb{N})(x : A)(xs : \mathit{Vec}\ A\ n)C\ n\ xs \to C\ (\mathit{succ}\ n)\ (\mathit{vcons}\ A\ n\ x\ xs)}{\mathbf{E}_V\ A\ C\ f_0\ f_1 : (m : \mathbb{N})(z : \mathit{Vec}\ A\ m)C\ m\ z}$$

$$\mathbf{E}_V\ A\ C\ f_0\ f_1\ \mathit{zero}\ (\mathit{vnil}\ A) = f_0$$

$$\mathbf{E}_V\ A\ C\ f_0\ f_1\ (\mathit{succ}\ n)\ (\mathit{vcons}\ A\ n\ x\ xs) = f_1\ n\ x\ xs\ (\mathbf{E}_L\ A\ C\ f_0\ f_1\ n\ xs)$$

2.1.1 Computation on inductive families

This section ends with some examples of definitions via elimination operators. These terms will be used in examples of coercion execution later.

- $plus =_{\mathrm{df}} [x, y : \mathbb{N}]\mathbf{E}_{\mathbb{N}}\ ([n : \mathbb{N}]\mathbb{N})\ y\ ([m : \mathbb{N}][p : \mathbb{N}]succ\ p)\ x : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$
- Summing up a list of numbers, $sum_L : List\ \mathbb{N} \to \mathbb{N}$, and $sum_L =_{\mathrm{df}} \mathbf{E}_L\ \mathbb{N}\ ([l : List\ \mathbb{N}]\mathbb{N})\ zero\ ([x : \mathbb{N}][l : List\ \mathbb{N}][s : \mathbb{N}]plus\ x\ s)$
- Converting a vector to a list, $v2l : (A : Type)(n : \mathbb{N})\, Vec\ A\ n \to List\ A$, where

$$v2l\ =_{\mathrm{df}}\ [A : Type][n : \mathbb{N}]\mathbf{E}_V\ A\ ([m : \mathbb{N}][v : Vec\ A\ m]List\ A)\ (nil\ A)\ ([m : \mathbb{N}][x : \mathbb{N}][l : Vec\ A\ m][t : List\ A]cons\ A\ x\ t)$$

- The opposite direction (a list to a vector) is less simple. We must provide size information, hence the definition of *length*. Term $l2v$ is often described as a *dependent coercion* [16], where the target type depends on the source value.

$$\begin{aligned} length : &\quad (A : Type)\ List\ A \to \mathbb{N} \\ &=_{\mathrm{df}}\ [A : Type]\ \mathbf{E}_L\ A\ ([l : List\ A]\mathbb{N})\ zero\ ([x : A][l : List\ A][t : \mathbb{N}]succ\ t) \\ l2v\ : &\quad (A : Type)\ (l : List\ A)\, Vec\ A\ (length\ A\ l) \\ &=_{\mathrm{df}}\ [A : Type]\ \mathbf{E}_L\ A\ ([l : List\ A]\, Vec\ A\ (length\ A\ l))\ (vnil\ A) \\ &\qquad ([x : A][t : List\ A]vcons\ A\ (length\ A\ t)\ x) \end{aligned}$$

2.2 Coercive subtyping

The wider conception of coercive subtyping in this paper derives from [13]. A coercion is a function $c : K_0 \to K$, which lifts an object of type K_0 to type K. The meaning of coercion use may be expressed via the *coercive definition rule* [13]:

$$\frac{f : (x : K)K' \quad k_0 : K_0 \quad K_0 <_c K}{f(k_0) = f(c(k_0)) : [c(k_0)/x]K'}$$

This says that term $f(k_0)$ abbreviates and is definitionally equal to a term where the coercion is made explicit, namely $f(c(k_0))$, when a coercion c exists to lift object k_0 to the type expected by the functional operation f. Notice that coercions are only used in a context where the expected type is known, i.e. where we know both K_0 and K. This paper does not consider coercions on higher types, e.g. contravariant sub-typing on dependent products.

Users can declare 'primitive' coercions, and 'derived' coercions can be synthesised by combining new coercions with old. The conventional transitivity rule generates $A <_h C$ with $h = g \circ f$ from $A <_f B$ and $B <_g C$. We also allow 'nesting' of coercions, e.g. lifting of coercions on element types to coercions over lists or vectors [6]. There are limits on what is allowed as a coercion: the resulting coercion set must satisfy *coherence*, the property that coercions between any two types are unique up to conversion. In any sizeable example, it is not unusual that several coercion terms may be derivable for a given source and target, particularly if arising from different

combinations of transitivity and nesting of coercions. Thus we must be sure that all possibilities lead to the same result. So-called "coherence checking" of a set of coercions is in principle undecidable, and thus an interesting question for practical implementations. Coherence is a major problem for parametrized coercions, which are essential for realistic (i.e., large-scale) use. A related problem is elimination of transitivity of coercion formation, to avoid unbounded search.

Forms of coercive subtyping have been implemented in Lego [3], Coq [19], Plastic [5], and Matita [2]. The coherence checking provided in [3] and [19] is decidable because both use a restricted form of coercion (based on syntactic matching of head type constructor). Matita's implementation is based on Coq though it supports a wider range of coercions by virtue of its more powerful handling of multiple inheritance via pullbacks in the coercion graph [18]. Plastic's implementation is more experimental: it allows full conversion tests to be used, and provides very powerful forms of coercion, but there are open problems to solve.

Several meta-theoretic results have been established over type theories such as UTT, subject to coherence. Elimination of transitivity in sub-typing has been proved for subsets of inductive types (and families) [9,10]. (The limitation is to non-recursive container types whose coercions are defined using projections rather than by direct elimination, and this enables various proofs to go through. The current work can be understood as a generalisation of this technique to recursive types.) Some work has explored weaker notions of transitivity [9]. Issues of structural subtyping in parametrized types are considered in [14]; proofs of coherence and transitivity are obtained by *extending* the underlying framework with new equalities that represent functorial properties of individual parametrized types, e.g. $map\ f \circ map\ g = map\ (f \circ g)$ for lists. More tentatively, efforts like "Observational type theory" [1], which carefully add forms of extensionality to intensional type theories, may also provide sufficient leverage for some of the problems discussed here.

Coercive subtyping generalises previous notions of subtyping. There are many applications, many of which provide useful abbreviations that ease use of complex constructions. The classic example is the use of coercions between levels of algebraic structure, where one may supply a value representing a group where (e.g.) a set is required: one just 'forgets' the additional structure. This facility has been much used in substantial formalization work. In the richer context of dependent type theory, coercions allow novel and interesting forms of subtyping, e.g. in representation of Natural Language lexical semantics [15], or in providing a bridge between easy-to-use simple types and more precise dependent types [6].

Coercions are also useful in programming with inductive families: coercions may be lifted functorially over inductive types, e.g. coercing a List of element type A to a List of element type B given a coercion $c : A \to B$ [13,4]. We are also finding interesting applications through regular use in Plastic. The parametrized coercion from function spaces (i.e., non-dependent Π-types) to the (dependent) Π-types is proving very useful: N-ary functions can be written in a simple notation but then coerced automatically to the required complex type. Subtyping also has applications with universes [5]. Such coercions help to simplify interfaces to proof tools.

2.3 Discussion

Coercive subtyping is potentially a powerful and useful framework. It has shown benefits in recent formalization work, and shows promise as a tool in user interfaces for proof systems and in programming with dependent types.

There are significant limitations, however. Current implementations require restrictions on coercions in order to guarantee key properties. For theory, some progress has been made beyond the implemented classes of coercion, but only for certain subsets and the strength of results is not uniform across the subsets. (A uniform treatment is arguably easier for users to understand, e.g. they don't have to learn a mixture of restrictions or analyze which ones might have applied when their abbreviation is rejected!) There are important classes of conversions for which no results have been established (e.g. between related container classes). It is important to make progress on these issues if coercive subtyping is to justify the claim of being a good abbreviation mechanism.

Why do these problems arise? The author's view is that the conventional presentation of Coercive Subtyping is too powerful, in the sense of not adequately limiting how it is used, and that many problems arise by trying to understand or to control the power at later stages. The conventional presentation also seems too divorced from how computation works on inductive families in an intensional theory.

One possibility is to find a more constrained presentation that provides a better balance between flexibility and ease of establishing key properties.

3 A different approach

3.1 A motivating example

Consider the coercion $v2l$ from vectors to lists (section 2.1.1) and its use when sum_L over lists is applied to vectors of numbers, e.g. in $sum_L \ < 1, 2, 3 >$. (For convenience, numerals denote equivalent values in the $\mathbb{N}$ type, and inferrable arguments are often omitted.) The conventional approach implies a conversion of the vector $< 1, 2, 3 >$ to a list, giving overheads of three new *cons* nodes to be allocated and time taken to do the extra work. This is wasteful, especially with the intuition that the folding could, in some sense, just traverse the vector structure and pick out the useful information. Can sum_V (sum on vectors) be defined in terms of how summation works on lists? The composition of sum_L with $v2l$ works, but a better (e.g. more efficient) version is possible when we have access to the arguments to the elimination operator inside sum_L. That is, when:

$$
\begin{aligned}
sum_L &=_{\text{df}} \mathbf{E}_L \ \mathbb{N} \ C \ zero \ f_1 \\
C &=_{\text{df}} [x : List \ \mathbb{N}]\mathbb{N} \\
f_1 &=_{\text{df}} [x : \mathbb{N}][xs : List \ \mathbb{N}][h : \mathbb{N}] plus \ x \ h
\end{aligned}
$$

then we can define sum_V as the following, assuming coercion function[3] $v2l$:

$$\begin{aligned} sum_V &=_{\text{df}} \mathbf{E}_V\ \mathbb{N}\ C'\ zero\ f_1' \\ C' &=_{\text{df}} [n : \mathbb{N}][v : Vec\ A\ n]C\ (v2l\ A\ n\ v) \\ f_1' &=_{\text{df}} [n : \mathbb{N}][x : A][v : Vec\ A\ n][h : C'\ n\ v]f_1\ x\ (v2l\ A\ n\ v)\ h \end{aligned}$$

Thus we have defined sum_V purely by modification of the arguments of the eliminator in sum_L. This sum_V is convertible with the direct definition on vectors. In fact, given arbitrary C, f_0, f_1, we can transfer any elimination on lists to one over vectors. We aim to use this kind of transformation as the basis for coercive subtyping.

3.2 Characterisation of applicable coercions

For which pairs of types can this transformation be done? And how is the transformation calculated? In this paper, we give a conservative answer, based on prior existence of a coercion function, i.e. a term generated by standard schema (e.g. for functorial maps [14]) *or* nominated by the user. (It may be possible to calculate the smallest function between the a given pair of types that unambiguously preserves appropriate information, if it exists. We leave this question for another paper.)

Given an existing coercion term, the criterion is that for each constructor of the target type, we can uniquely determine how it maps to source type constructors, and hence how the relevant elimination argument for the target type constructors should be transformed. This covers all functions whose injectivity is decidable, and so encompasses functorial maps and many of the natural transformations between 'similar' datatypes (similar in the sense that none of the core data content is lost). It does not, however, include terms like first projection on dependent pairs (Σ-types) – see section 3.8. We also have to transform the *motive* (denoted C, C' in the examples); this is more straightforward and can be determined by inspection of the family indices of the types concerned.

As a first attempt, the transformations will be expressed as a particular fixed form of eliminator: the parameters, motive and branches (i.e. functions for each constructor) will be based on the target type, but indices and the value to eliminate over will be based on the source type. The head symbol of the function body will also be the eliminator of the source type. Defining this as a well-typed term also guarantees the type correctness of a transformation. For the vector to list example, this means a term $\mathbf{E}_{LV}$ as defined below. (Such terms are henceforth named $\mathbf{E}_{XY}$, where X is the target type and Y the source type.)

$$\frac{C : List\ A \to Type \quad f_0 : C\ (nil_A) \quad f_1 : (a : A)(l : List\ A)C\ l \to C\ (cons_A\ a\ l)}{\mathbf{E}_{LV}\ A\ C\ f_0\ f_1 : (n : \mathbb{N})(v : Vec\ A\ n)C\ (co_V\ A\ n\ v)}$$

[3] It appears here that we still retain the coercions, but this is informal notation representing the coercion of sub-values via the new mechanism, and will be made more precise later. Notice that in this example, both C and f_1 will discard the coerced terms passed to them. This is true for all 'iterative' or folding computations (as opposed to primitive recursive ones).

$$\mathbf{E}_{LV}\ A\ C\ f_0\ f_1 =_{\mathrm{df}} \mathbf{E}_V\ A\ ([n:\mathbb{N}][v:\mathit{Vec}\ A\ n]C\ (\mathit{co}_V\ A\ n\ v))\ f_0$$
$$([m:\mathbb{N}][x:A][v:\mathit{Vec}\ A\ m][H:C\ (\mathit{co}_V\ A\ m\ v)]$$
$$f_1\ x\ (\mathit{co}_V\ A\ m\ v)\ H)$$

The term contains occurrences of symbol co_V in places where where conversions from vectors to lists may still be required, specifically in the motive's main argument and in branch functions where recursive sub-values must be converted. For now, co_V should be read as a constructor rather than as a function: the reasons for this are explained in section 3.3.

The definition of $\mathbf{E}_{LV}$ is not a case of shifting a problem elsewhere or of introducing circularity. Firstly, it is correct wrt. type checking – that's the result type we must expect since C etc are not yet known. Secondly, consider the three possible futures for each occurrence: (a) the coerced value is discarded without examining it; (b) some elimination is performed on it; or (c) no computation occurs, e.g. in the case of C being a type constructor or a variable. Cases (a) and (b) are certainly fine, since either way the coerced term will eventually disappear. Note that case (a) tends to be more frequent, i.e. C doesn't often examine the elimination target. For case (b), the elimination will be over lists, and the coerced (sub-)term is handled in the same way as the original term. In case (c), the blocked term will only be tested with conversion. The interaction with conversion is considered later.

Observe that $\mathbf{E}_{LV}$ precisely explains how to transform computations on lists to computations on vectors, and does so in a way that is central to the meaning of the types – via their eliminators. This statement of coercibility is arguably more meaningful than just providing some function to convert from one to the other, in the sense that it shows *why* and *how* coercion is possible. Such structure is very useful in establishing proofs of relevant properties (section 3.5), compared to the conventional conversions which are effectively opaque and unanalysable in an intensional setting. The proposal is to use the $\mathbf{E}_{XY}$ terms as the basis of coercive subtyping.

Currently, terms like $\mathbf{E}_{LV}$ are defined manually by inspection of the conventional coercion terms (e.g. from $v2l$). Automatic derivation by inversion of such terms or by analysis of the inductive schemata of two types is planned as further work.

3.3 Modifying the ι-reduction mechanism

We now consider how these transformations are used. Firstly, instead of inserting a coercion function when a type mismatch is found, we insert a marker representing a coerced term. Currently, the marker bears a label for the source type and the parameters and indices such that the resulting term is well-typed, e.g. a coerced vector v is represented as $(\mathit{co}_V\ A\ n\ v)$, where the label is assigned the type $(A : \mathit{Type})(n : \mathbb{N})\,\mathit{Vec}\ A\ n \to \mathit{List}\ A$. Note that well-typedness ensures that the target type is known and does not need to be labelled explicitly. This constant wrapper *delays action* on the coerced value until we know what computation is to be performed on it. To 'activate' the coercion, the marker must reach an elimination

operator of the target type, at which point the appropriate $\mathbf{E}_{XY}$ term is used. We represent this by adding a further computation rule for Lists:

$$\mathbf{E}_L\ A\ C\ f_0\ f_1\ \ (co_V\ A\ n\ v) \mapsto \mathbf{E}_{LV} A\ C\ f_0\ f_1\ n\ v$$

Should other coercion transformations be required (subject to conditions, e.g. that a suitable $\mathbf{E}_{XY}$ term can be defined and that coherence is preserved), they will each result in an additional computation rule in a form similar to the last line. Note that the form of the new reductions reflects those for decoding Tarski-style universes [5], e.g. $\mathbf{T}_{i+1}(\mathbf{t}_{i+1}(a)) = \mathbf{T}_i(a) : \mathit{Type}$, where decoding of a lifted name is handled by decoding the unlifted name.

3.4 Canonicity and interaction with conversion

There are two aspects to canonicity: first, what happens when coerced values are compared with other values in the target type; and second, the extent to which the coercion markers act as constructors for the target type.

The first situation concerns questions such as the convertibility of a list with a vector value coerced to a list, i.e. $n : \mathbb{N}, v : \mathit{Vec}\ A\ n, l : \mathit{List}\ A \vdash (co_V\ A\ n\ v) = l$, or convertibility of values coerced from two different source types into the same target type. In the standard approach, (a) the coercion function performs the transformation on non-blocked terms and convertibility can proceed; or (b) the term is blocked, e.g. as a variable, so the coercions remain in place and conversion fails unless the coerced values are convertible.

It is different in the new scheme, because the transformation is implicit in some elimination, and convertibility (by itself) does not add eliminations. But there is a computation that can be applied safely to a term to remove the coercion at its head: the *identity elimination* for the target type. Note that this does not imply significant overheads for convertibility: in a sense, it performs a conversion that was delayed from an earlier time, and performs work that is identical to the conventional transformation function. [4] In cases where the reduction is blocked, we are in no worse a situation than the same case in the original scheme.

To clarify, the proposal is add reductions (not equivalences) of the form below, where $C_{Id(X)}$ is the identity elimination motive for inductive type X and $\overline{f_{Id(X)}}$ the identity branch functions for X. This form is necessary to ensure that the coercion marker is removed. Such reductions will only apply to coerced terms. Note that such reductions are similar to η rules on inductive types, and that η rules are not problem-free [12, p. 198], but the limitation to coercions and only to complete delayed coercions may prove sufficient. (This will be investigated as further work.)

$$co_Y\ \overline{p}\ \overline{q}\ y \mapsto \mathbf{E}_{XY}\ \overline{p}\ C_{Id(X)}\ \overline{f_{Id(X)}}\ \overline{q}\ y$$

The second situation is less clear. The question is whether and to what extent coerced terms, particularly blocked ones (which can't progress with computation),

[4] That is, $\mathbf{E}_{LV}\ A\ C_{Id(L)}\ \overline{f_{Id(L)}}\ n\ v = v2l\ A\ n\ v$, and this is easily proved. This property shows that the relevant $\mathbf{E}_{XY}$ term was correctly constructed, so it is a useful check to make.

are to be taken as canonical values of the target type. Subtyping does blur the notion of canonicity. An answer to this question has bearing on how reductions of the form above are used. We leave this issue for further work.

A related question is how much of the coercion process is revealed to the user, particularly in the context of a proof assistant where intermediate terms may be partially computed. Do we show details of the source type, of the target type, or a mixture? What is suitable for helping the user understand the state of his or her proof? One possibility is that reduction of $\mathbf{E}_{XY}$ should be extended to compute out the $\mathbf{E}_Y$ portions, i.e. to extract data from the source value and show a term containing original branch functions and extracted data. For example, $sum_L < 1, 2, 3 >$ could be reduced directly to something like $plus\ 1\ (plus\ 2\ (plus\ 3\ 0))$. The labelling framework from Epigram [17] may also help: it is designed to give clear information about computations at a level above raw eliminators. We also remark on the parallels between *views* in Epigram and the source-target relationship in coercions, in that one can provide coercions from a source type to its view types.

3.5 Compositions of $\mathbf{E}_{XY}$ *terms and transitivity*

We now consider how the $\mathbf{E}_{XY}$ terms behave under composition. The transitivity rule of coercive subtyping forms new coercions by composition, and coherence requires a check that new coercions are consistent with existing coercions. A related case is where values are coerced to one type and in the course of later computation are coerced again, possibly back to the original type, that is where a term $co_Y\ \overline{p'}\ \overline{q'}\ (co_Z\ \overline{p''}\ \overline{q''}\ v)$ is formed ($\overline{p}$ and $\overline{q}$ represent parameters and indices).

As a concrete example, consider the coercions between lists and vectors, and the cases of list-vector-list and vector-list-vector. Intuitively, the result should be an identity operation on lists (resp. vectors). In general, such "identity coercions" are excluded in coercive subtyping. One practical reason is that the resulting composition is in general not intensionally equal to identity, meaning that we would have to choose one direction (X to Y) or the other, but can not have both. We will examine the two combinations of $\mathbf{E}_{LV}$ (defined earlier) and $\mathbf{E}_{VL}$ (defined below).

$$\frac{\begin{array}{l} A : Type \quad C : (n : \mathbb{N})\, Vec\ A\ n \to Type \quad f_0 : C\ zero\ (vnil\ A\ zero) \\ f_1 : (m : \mathbb{N})(a : A)(v : Vec\ A\ m) C\ m\ v \to C\ (succ\ m)\ (vcons\ A\ m\ a\ v)) \end{array}}{\mathbf{E}_{VL}\ A\ C\ f_0\ f_1 : (z : List\ A) C\ (length\ A\ z)\ (co_L\ A\ l)}$$

$$\mathbf{E}_{VL}\ A\ C\ f_0\ f_1 =_{\mathrm{df}} \mathbf{E}_L\ A\ ([l : List\ A] C\ (length\ A\ l)\ (co_L\ A\ l))\ f_0\ ([x : A][t : List\ A] f_1\ (length\ A\ t)\ x\ (co_V\ A\ m\ v))$$

We consider the composition of $\mathbf{E}_{LV}$ with $\mathbf{E}_{VL}$ under arbitrary A, C, f_0, f_1 and a coerced list value l. It shows the effect of two coercions in sequence. The composition is not direct, i.e. does not take the usual form $f\ \overline{a} \circ g\ \overline{b}$, since we will require reduction of $\mathbf{E}_{LV}$ to compute changed arguments for $\mathbf{E}_{VL}$. In the third line, $\mathbf{E}_{VL}$ is 'activated'

to transfer the computation on vectors to one on the (coerced) list.

$$
\begin{aligned}
&\mathbf{E}_{LV}\ A\ C\ f_0\ f_1\ (\mathit{length}\ A\ l)\ (\mathit{co}_L\ A\ l)\\
=_{\delta\beta}\ &\mathbf{E}_V\ A\ ([n:\mathbb{N}][v:\mathit{Vec}\ A\ n]C\ (\mathit{co}_V\ A\ n\ v))\ f_0\\
&\qquad([m:\mathbb{N}][a:A][v:\mathit{Vec}\ A\ m][H:C\ (\mathit{co}_V\ A\ m\ v)]f_1\ a\ (\mathit{co}_V\ A\ m\ v)\ H)\\
&\qquad(\mathit{co}_L\ A\ l)\\
=_{co}\ &\mathbf{E}_{VL}\ A\ ([n:\mathbb{N}][v:\mathit{Vec}\ A\ n]C\ (\mathit{co}_V\ A\ n\ v))\ f_0\\
&\qquad([m:\mathbb{N}][a:A][v:\mathit{Vec}\ A\ m][H:C\ (\mathit{co}_V\ A\ m\ v)]f_1\ a\ (\mathit{co}_V\ A\ m\ v)\ H)\ l\\
=_{\delta\beta}\ &\mathbf{E}_L\ A\ ([k:\mathit{List}A]C\ (\mathit{co}_V\ A\ (\mathit{length}\ A\ k)\ (\mathit{co}_L\ A\ k)))\ f_0\\
&\qquad([a:A][k:\mathit{List}A][H:C\ (\mathit{co}_V\ A\ (\mathit{length}\ A\ k)\ (\mathit{co}_L\ A\ k))]\\
&\qquad\quad f_1\ a\ (\mathit{co}_V\ A\ (\mathit{length}\ A\ k)\ (\mathit{co}_L\ A\ k)))\\
&\qquad l
\end{aligned}
$$

We require that this computation is identical to that done directly on l, so compare it with term $\mathbf{E}_L\ A\ C\ f_0\ f_1\ l$. There are disagreements in the motive and step-case function, shown here as required conversions (in η-long form):

$$
\begin{aligned}
C &= [k:\mathit{List}\ A]C\ (\mathit{co}_V\ A\ (\mathit{length}\ A\ k)\ (\mathit{co}_L\ A\ k))\\
f_1 &= [a:A][k:\mathit{List}A][H:C\ (\mathit{co}_V\ A\ (\mathit{length}\ A\ k)\ (\mathit{co}_L\ A\ k))]\\
&\qquad f_1\ a(\mathit{co}_V\ A\ (\mathit{length}\ A\ k)\ (\mathit{co}_L\ A\ k))\ H
\end{aligned}
$$

The problem here is the equation $k = \mathit{co}_V\ A\ (\mathit{length}\ A\ k)\ (\mathit{co}_L\ A\ k)$, i.e. whether the two coercions together correspond to an identity. However, note that co_V and co_L are constructors and they only have meaning when some elimination is applied. Section 3.3 suggested application of the relevant identity elimination. This still leaves the problem of $k = \mathbf{E}_L\ A\ C_{Id(X)}\ \overline{f_{Id(X)}}\ k$, i.e. of whether the identity elimination really is an identity elimination. For now, we take it as a premiss of the proposition and conclude this: if the identity elimination(s) are identities for conversion, then we can prove coherence of the composed coercions.

The other direction is less straightforward, from vectors to lists to vectors as the composition of $\mathbf{E}_{VL}$ with $\mathbf{E}_{LV}$. Under arbitrary A, C, f_0, f_1, and coerced vector v of size n, the critical equations are:

$$
\begin{aligned}
C\ n\ v\ \ &= C\ (\mathit{length}\ A\ (\mathit{co}_V\ A\ n\ v))\ \ (\mathit{co}_L\ A\ (\mathit{co}_V\ A\ n\ v))\\
f_1\ m\ x\ v\ &= f_1\ (\mathit{length}\ A\ (\mathit{co}_V\ A\ m\ v))\ x\ (\mathit{co}_L\ A\ (\mathit{co}_V\ A\ m\ v))
\end{aligned}
$$

Assuming the properties of identity eliminations suggested above, convertibility is blocked here only by the vector length component, i.e. $(\mathit{length}\ A\ (\mathit{co}_V\ A\ n\ v)) = n$ is not derivable intensionally. It can be proved extensionally by induction over n or v. Note that this is still a computation applied to a coercion, but reduction via

$\mathbf{E}_{LV}$ does not help: the $\mathbf{E}_V$ elimination can not be reduced further. [5]

To summarise, a composition of $\mathbf{E}_{LV}$ with $\mathbf{E}_{VL}$ is intensionally equal to the uncoerced computation under assumption of properties of identity eliminations. The opposite direction is blocked by an equation on the vector's size parameter. We suggest that coherence is provable in a similar way for a wider range of types for which $\mathbf{E}_{XY}$ terms can be defined: (a) all inductive types (i.e. no family indices); (b) inductive families with non-recursive indices; (c) inductive families where the $\mathbf{E}_{XY}$ terms do not exhibit dependencies across the result of composition. The justification is that the $\mathbf{E}_{XY}$ terms avoid recursion on intermediate values and the critical terms to check (arguments to the elimination operators) are convertible by virtue of how the $\mathbf{E}_{XY}$ are constructed. Note that this conjecture covers both compositions of coercions within the same type (including the functorial coercions studied in [14]), AND the more general compositions involving two or three different types (on which no work has yet been done). The issues of inductive family parameter behaviour in composed coercions requires further study.

The value of such a result (when formally proved) is to provide an intensional and simple to automate method to check coherence of a wider range of coercion combinations, particularly those arising from transitive closure of coercions.

3.6 Preliminary remarks on metatheory

The mechanism proposed in section 3.3 is a modification of how coercions are specified and of their reduction behaviour, with subsequent effects on how key properties are stated. The modification exposes details of how computation on one type is mapped to computation on the other, and delays the action of 'coercion' until it is known what computation to apply (else applies an identity elimination).

The new ι-reductions are sound wrt. types because of the fixed form of the $\mathbf{E}_{XY}$ terms and the way in which they are used in ι-reductions. (Indeed, preliminary experiments have been developed and checked in Plastic [5].) Correctness of the definition of $\mathbf{E}_{XY}$ terms can be stated by comparison with the conventional conversion function (possibly from which the definition has been extracted), and easily proved by applying the identity elimination. That is, the $\mathbf{E}_{XY}$ term is correct if the original conversion function is correct. (In future, we may require $\mathbf{E}_{XY}$ terms to be defined from the structure of the relevant inductive schemata, which will give a stronger guarantee of correctness.)

Use of the default identity elimination in conversion appears safe, in the sense that redexes arise because of a delayed computation and the reduction effectively executes the conversion of representation originally specified by the user (again, this gives rise to a proof obligation which is easily proved by induction). This redex is thus equivalent to the situation in the standard approach of forcing reduction on an implicitly coerced value, and relevant results from the literature should apply.

[5] It may possible to get round this particular case by introducing a coercion from vectors to their lengths, i.e. to express computation on $\mathbb{N}$ in terms of a vector traversal and thus avoid the elimination on vectors, but this is hardly a general solution.

3.7 Implementation details, and efficiency

Preliminary experiments have been carried out in Plastic [5], a system which implements Luo's LF with Coercive Subtyping. Several $\mathbf{E}_{XY}$ terms have been defined and the additional ι-reductions (of eliminators) simulated via 'back door' access to the inductive families implementation. (This back door allows non-standard inductive types to be defined manually. The proposed reductions are checked for type safety, but termination is not checked.) Relevant proofs have been developed in Plastic, where feasible, or the reasons for failure analysed.

The full mechanism, including identity elimination reductions in conversion, will be straightforward to implement. Identity eliminators can be derived from schemata. The types of $\mathbf{E}_{XY}$ terms can be generated automatically from the relevant schemata. Derivation of definitions of $\mathbf{E}_{XY}$ terms may be possible for simple cases, else the user can develop the definition by refinement. The existing algorithms for generating and matching coercions will not need changes.

Improvements in efficiency of coercion execution are expected. Firstly, there is no intermediate structure to be built and then traversed: computations are applied directly to the original data (this technique has parallels with deforestation in functional programming). Secondly, the $\mathbf{E}_{XY}$ computations can be improved in several ways, not least some form of partial evaluation or Normalization by Evaluation on the branch functions to avoid repeated work later, or the collapsing of chains of coercions (e.g. projections on algebraic structures) to simpler functions.

3.8 Further examples

We briefly consider two different examples. Firstly, projections from Σ-types. Some authors suggest π_1 as a useful coercion, though one recent study identifies problematic interactions of this coercion with functorial mappings on Σ [9]. That work suggests a two-stage application of coercions, effectively constraining how these two groups of coercion can be composed: π_1 is used only in the second stage, after all other applicable coercions have been inserted.

$$\Sigma : (A : \mathit{Type})(B : A \to \mathit{Type})\,\mathit{Type}$$

$$\pi_1 : (A : \mathit{Type})(B : A \to \mathit{Type})\Sigma\ A\ B \qquad \to A$$

$$\pi_2 : (A : \mathit{Type})(B : A \to \mathit{Type})(s : \Sigma\ A\ B) \to B\ (\pi_1\ A\ B\ s)$$

In the new framework, can π_1 be expressed as a coercion? The first question is whether (and how) we can transfer computation on A to computation on the Σ value. Nothing is known about A, hence the answer is negative and we reject π_1 as a coercion: it makes no sense.[6] Note that the functorial mappings on Σ *are* expressible as $\mathbf{E}_{XY}$-style terms (see below for a similar example).

[6] It may be interesting to consider a weaker conversion term, say of type $(A \to C) \to \Sigma\ A\ B \to C$ which reflects that A will be transformed to C. Investigating such terms and the possibility of integration with the main mechanism is planned as further work.

The second example involves lifting a coercion over a container data type, specifically lifting a coercion on element types to a coercion on lists of that element. Such a rule can be applied recursively, hence used to convert arbitrarily nested lists. (Such recursive coercions have been implemented in Plastic for the conventional approach [6,4].) The basic form of $\mathbf{E}_{LL}$ is given below, expressing computation on $List\ A$ values in terms of $List\ B$ computations. Notice that two extra parameters are needed: the type of the source list elements and a coercion function on the elements. The precise representation of such 'nested' coercions has not been decided; for now, the simplest representation is chosen. The key detail below is that the modified step-case function applies the element conversion function to the head element before proceeding. Relevant coherence properties hold intensionally for this term, similarly to the $\mathbf{E}_{LV} - \mathbf{E}_{VL}$ composition. Observe that $\mathbf{E}_{LL}$ pre-composes with $\mathbf{E}_{LV}$ etc.

$$\begin{aligned}
\mathbf{E}_{LL} : \ & (B : Type)(C : [l : List\ B]Type)(f_0 : C\ nil) \\
& (f_1 : (x : B)(xs : List\ B)(H : C\ xs)C\ (cons\ B\ x\ xs)) \\
& (A : Type)(f : A \to B)(l : List\ A)C\ (co_{LL}\ A\ B\ f\ l) \\
\mathbf{E}_{LL} = \mathbf{E}_L\ & A([l : List\ A]C\ (co_{LL}\ A\ B\ f\ l))\ f_0 \\
& ([x : A][xs : List\ A]f_1\ (f\ x)\ (co_{LL}\ A\ B\ f\ xs))
\end{aligned}$$

3.9 Discussion and future work

This work is still in early stages, but early results are promising and there are several interesting extensions to pursue. The work contributes in several ways: (a) characterising an important subset of coercion functions; (b) enabling proof of key properties of this subset; (c) supporting greater efficiency of coercion use. Progress has been made towards simpler proofs on functorial coercions, and towards new proofs for more general cases of coercion combination (e.g. compositions involving several distinct types) that were previously identified as problematic. All of these aspects are important to promote Coercive Subtyping as a useful and practical abbreviation mechanism.

Future work includes formal proofs, full implementation in Plastic to enable larger studies, and further study on the class of $\mathbf{E}_{XY}$ terms - including automatic derivation and the study of restrictions (e.g. indices of inductive families). Coercions that extend compatibility of inductive family indices appear particularly interesting, e.g. $Eq\ m\ n \to Vec\ A\ m \to Vec\ A\ n$.

Acknowledgement

Thanks to the referees and Zhaohui Luo for their interesting and useful comments.

References

[1] Altenkirch, T. and C. McBride, *Towards observational type theory*, Manuscript, available online (2006).

[2] Asperti, A., C. Sacerdoti Coen, E. Tassi and S.Zacchiroli, *Crafting a proof assistant* (2006), submitted for publication to the TYPES06 Post Proceedings.

[3] Bailey, A., "The Machine-checked Literate Formalisation of Algebra in Type Theory," Ph.D. thesis, University of Manchester (1998).

[4] Callaghan, P., *Coercions in Plastic*, Lecture notes, TYPES Summer School 2002 (2002), http://www-sop.inria.fr/certilab/types-sum-school02/.

[5] Callaghan, P. and Z. Luo, *An Implementation of LF with Coercive Subtyping & Universes*, Journal of Automated Reasoning (Special Issue on Logical Frameworks) **27** (2001), pp. 3–27.

[6] Callaghan, P., Z. Luo and J. Pang, *Object languages in a type-theoretic meta-framework*, in: *Proc. Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs (PTP'01)*, 2001.

[7] Coquand, T. and C. Paulin-Mohring, *Inductively defined types*, LNCS **417** (1990).

[8] Dybjer, P., *Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics*, in: G. Huet and G. Plotkin, editors, *Logical Frameworks* (1991).

[9] Luo, Y., "Coherence and Transitivity in Coercive Subtyping," Ph.D. thesis, U. of Durham (2004).

[10] Luo, Y. and Z. Luo, *Coherence and transitivity in coercive subtyping*, in: *Proc. 8th Intl. Conf. on Logic for Programming, Artificial Intelligence, and Reasoning*, 2001, pp. 249–265, (LNAI 2250).

[11] Luo, Z., *A unifying theory of dependent types: the schematic approach*, Proc. Symposium on Logical Foundations of Computer Science (Logic at Tver'92), LNCS 620 (1992).

[12] Luo, Z., "Computation and Reasoning: A Type Theory for Computer Science," OUP, 1994.

[13] Luo, Z., *Coercive subtyping*, Journal of Logic and Computation **9** (1999), pp. 105–130.

[14] Luo, Z. and R. Adams, *Structural subtyping for inductive types with functorial equality rules*, Math. Struct. in Comp. Science (2007), (to appear).

[15] Luo, Z. and P. Callaghan, *Coercive subtyping and lexical semantics (ext'd abstr.)*, LACL'98 (1998).

[16] Luo, Z. and S. Soloviev, *Dependent coercions*, The 8th Inter. Conf. on Category Theory and Computer Science (CTCS'99), Edinburgh, Scotland. Electronic Notes in Theoretical Computer Science **29** (1999).

[17] McBride, C. and J. McKinna, *The view from the left*, J. Functional Programming **14** (2004), pp. 69–111.

[18] Sacerdoti Coen, C., *A Presentation of Matita* (2006), slides from talk at CHIT/CHAT Workshop.

[19] Saïbi, A., "Outils Génériques de Modélisation et de Démonstration pour la Formalisation des Mathématiques en Théorie des Types. Application à la Théorie des Catégories," Ph.D. thesis (1999).