RC 21615 (log 97460) 1Dec99 Computer Science/Mathematics

IBM Research Report

M-types and Their Coercions

Charles M. Barton

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Research Division
Almaden · T.J. Watson · Tokyo · Zurich · Austin

This report discusses a system of recursive terms with n-ary sums and products, and provides a semantic model (M-types) of these terms. It defines a language of recursive coercions between these terms and an interpretation of coercions as partial functions between M-types. In this model "axioms" such as the associative, commutative, and distributive laws denote natural transformations between *polynomial functors*. These laws give isomorphisms but are not identities of M-types. An inference system is used to assign Curry-style types ($\tau \to \sigma$) to coercions, and we give conditions under which the interpretation of such a coercion is a total function $[\![\tau]\!] \to [\![\sigma]\!]$. Assuming the natural transformations and leaf coercions that occur in a coercion c denote injective (resp. surjective/bijective) maps, we show the same for c. We also show how to invert formally a bijective coercion.

1 Introduction

The problems considered in this report grew out of the Mockingbird project [4], which has developed a tool to generate adapters between programs written in possibly different programming languages and running in a distributed environment. The tool analyzes a pair of type declarations in the two languages and generates stub code to convert values of one type to values of the other type. These stubs are transparently invoked at runtime to allow interoperation between the programs. For example, when a function in one program is invoked from the other, its arguments are "coerced" to equivalent values in the other language, and the return value is coerced in the opposite direction.

Mockingbird was the work of Joshua Auerbach, Mark Chu-Carroll, Mukund Ragavachari, and the author. Building the tool involved considerable work in the runtime support for distributed computation and the user interface, as well as in more theoretical questions. The original strategy of translating type declaration to an internal form and matching these comes from a paper of Auerbach and Russell [6]. Not all aspects of the project have been presented but a series of references is available ([2, 3, 4, 5]). This report provides a solution to one of the theoretical questions raised by the project. Some of the remaining theoretical issues are discussed later (Section 9).

The equivalences found by the tool are based on structure rather than on names. Starting from identifications between primitive types, for example integers or characters, aggregate types such as records can be matched, and then recursive structures such as linked lists of records are made to correspond. Programmer annotations are used to assert constraints on a program's use of particular types, such as range limits for an integral type or assertions that pointers of a particular type will never be null. Annotations also allow the user to specify a particular correspondence in ambiguous cases, such as between records each having more than one field of a given primitive type. These annotations in effect refine the type systems as seen by the Mockingbird analyzer.

The problem of finding equivalences is complicated due to recursion and to the use of restructuring. First, recursion permits alternative declarations of types which any programmer would find equivalent. The C declaration

```
struct L {
   int m;
   struct L  *next;
};
```

describes a type whose values are linked lists of one or more integers (assuming the pointer "next" is permitted to be null). One can instead unfold the declaration by introducing separate types for the even and odd occurrences:

```
struct Even;
struct Odd {
   int n;
   struct Even *link1;
};
struct Even {
   int k;
   struct Odd  *link2;
};
```

Again assuming that link1 and link2 may be null, struct L, struct Odd, and struct Even should be considered equivalent. Techniques for detecting such equivalences go way back. Recent work of Amadio-Cardelli [1] and Brandt-Hengelein [7] has extended these techniques to provide an axiomatization of both equality and subtype relations.

A second difficult aspect of detecting type equivalence concerns the use of restructuring transformations such as reordering of fields in records. Considering formation of record types as analogous to multiplication, reordering of fields corresponds to applying the commutative law, and the use of nested records raises the question of the associative law. Similarly one can view formation of discriminated unions (resp function types) as analogous to addition (resp. exponentiation). This line of investigation leads to interesting questions of axiomatizability and decidability (see [9]).

The problems addressed by this report concern the definition and soundness of restructuring transformations in the presense of recursion. Before discussing this further, we explain the overall strategy used by the Mockingbird tool and introduce some notation.

In the Mockingbird project we refer to the set of values defined by a (possibly annotated) type declaration, in a supported programming language or IDL, as an S-type

("S" for "Surface"). We approach the problem of converting between two S-types by translating each declaration into an internal form, an M-term. Acceptable transformations between surface types are represented by an axiomatic system for deducing coercions between M-terms. These include both "algebraic" rules permitting for example reordering factors of a product as well as a generalization of the coinductive rule introduced in [7]. The latter allows one to introduce recursive coercions.

This approach raises questions of definition and of soundness. First, one may ask how formal manipulations of M-terms help to define transformations between S-types. We reply by interpreting M-terms as sets, called M-types, of abstract values and by interpreting the formal manipulations of M-terms as (partial) functions between M-types. The values are abstract in that compiler- and platform-specific details, such as endianness, packing of fields, and memory addresses, are ignored, and only the structural relations are retained. Each value of an S-type gives rise to a value of the corresponding M-type, and each M-type value can be so realized. The transformation of S-types is then logically a composition (though not implemented that way): given a value of the source S-type, take its abstraction, apply the function between M-types, and realize the resulting M-type value as a value of the target S-type. For a complete account of the transformation process, we would need semantic models for each of the supported surface languages and a mathematical description of the value mapping from S-types to M-types. This report only discusses the internal system of M-type and their coercions.

There are at least two questions of soundness: (1) whether the coercion functions between M-types always converge, and (2) whether these functions are injective/bijective/.... (1) arises when the surface types are recursive because the coercion functions are also recursive. We discuss this question further below. The answer to (2) depends on the repertory of primitive coercions and restructuring transformations used in building a coercion. We show (Section 7) that, given convergence, our compound coercions are injective/bijective/... when their components are, and that in the bijective case, the coercions can be inverted formally. This fact is important in guaranteeing that "round-trip" transformations of surface types are essentially identities, i.e., that when we transform a surface value v_1 of type T_1 to v_2 of type T_2 and then back to w of type T_1, that v_1 and w are "the same" from the point of view of the source language.

The question (2) of convergence is a real one, since undisciplined combinations of coercions may in fact diverge. We learned of this possibility from Roberto Di Cosmo [8] who pointed out the inconsistency of the axiom system obtained by extending the Amadio-Cardelli axioms (with functions replaced by products) by adding the algebraic axiom $A\times 1 = A$. Auerbach then found the same problem in a similar extension based on Brandt-Henglein axioms. Since the method used in the Mockingbird tool to discover coercions can be viewed as a search for a proof of coercibility, it is not surprising that unfortunate coercions can arise. One of the chief contributions of this report is to describe a class of *constrained* coercions for which convergence is assured. The class is described by a simple syntactic constraint which is sufficient for our purposes, since the coercions discovered by the Mockingbird tool satisfy the constraint. We do not have a necessary and sufficient condition for convergence. We give in Section 8 an example of a coercion which is unconstrained but nevertheless defines an isomorphism between the source and destination M-types.

The work reported here has been greatly influenced by [7], which introduced the formal derivation of coercions. A significant difference concerns the semantic model. Brandt-Henglein used the same language to denote types as did Amadio-Cardelli, allowing recursion and function types, based on constants for the top and bottom types. Type expressions have forms[1] like $(\mu\alpha.\sigma\rightarrow\alpha)$, which denotes a type T satisfying the equation $\mathsf{T} = \sigma \rightarrow \mathsf{T}$. This type can be "unfolded" to yield $\mathsf{T} = \sigma \rightarrow(\sigma \rightarrow \mathsf{T})$, and the unfolding can be repeated indefinitely. This has led to the semantic model of such types as possibly infinite regular trees ([1, 7, 10]), supporting sound and complete axiomatizations for equality and for the subtype relation, as well as decision algorithms.

In contrast, we take the meaning of an M-term to be a set of values, which are terms whose forms are derived from the structure of the M-term. (This value set can also be derived from the infinite regular tree.) These values are apt models for the surface values, and an additional advantage of this approach appears when we consider M-terms related by algebraic transformations. Consider for example $\mu\ \alpha.(\mathsf{A} + \alpha\times\mathsf{B})$ and $\mu\ \alpha.(\mathsf{A} + \mathsf{B}\times\alpha)$. The infinite trees they denote are certainly unequal, as are their M-types, but in our case the algebraic transformation induces a bijection between the M-types. This leads us to focus on coercibility rather than equality, and on the existence of injective coercions rather than on subtyping. Since M-types and coercions are just sets and functions, it is natural to consider the collection of all M-types and their coercions as a category. Then expressions such as $\mathsf{y}_1\times(y_2+\mathsf{y}_3)$ are observed to define functors on this category, and the algebraic laws such as distributivity appear as natural transformations between these functors. Such natural transformations are explicitly represented in formal coercions, and their properties are used in the proof of termination.

Another unfortunate difference between our work and that cited above is the lack of a decision procedure. We have no proof of undecidability, but unpublished work of Joseph Gil [11], in a related setting, suggests at least the impossibility of deciding whether there is an *injective* coercion between two M-terms. As a practical matter in the Mockingbird project, this seems relatively unimportant, since our goal is to find translation intuited by programmers. Once the type declarations have been suitably annotated, it seems likely

[1] $\mu\alpha$ introduces a recursion. See Section 2

that the coercions will be found by a rather shallow search. Another alternative is to provide for proof hints by the user. These issues are discussed further in Section 4.3.

In the next section we define the language of M-terms and its model, the cpo $\mathcal{M}$ whose elements are the M-types. We describe the semantic function, which maps a closed M-term τ to its M-type $[\![\tau]\!]$. We introduce a running example of two M-terms translated from C and Java type declarations, which we will revisit in later sections.

In Section 3 we use the fact that $\mathcal{M}$ the cpo of all M-types, is also the set of objects of a category. We study a class of functors $\mathcal{M}^p \rightarrow \mathcal{M}^q$ which we called "polynomial" because in fact they are given by terms built from variables and type constants using only n-ary sums and products. These functors correspond to the left and right sides of algebraic axioms like the distributive law. The equality asserted by such axioms is replaced by a natural transformation between such functors.

In Section 4 we define a language of coercions, recursive terms involving (names for) natural transformations. These are used to combine restructuring with the effects of subordinate coercions. We also employ a system of rules and axioms, adapted from [7], for assigning types to coercions. A coercion type has the form $(\sigma \rightarrow \tau)$, where σ and τ are closed M-terms, and we are interested in showing that the function denoted by c maps $[\![\sigma]\!]$ to $[\![\tau]\!]$. The semantic mapping from coercion terms to partial functions on M-types is given in Section 5.

The question of termination is treated in Section 6. We define the class of constrained coercions and show that they have the desired behavior. The constraints are used in an inductive proof showing that recursive invocations of coercions receive "smaller" arguments on each call. Invertibility is discussed in Section 7. In Section 8 we presents a well-typed but unconstrained coercion which nevertheless is a bijection between its source and destination M-types. In Section 9 we indicate several open questions.

Three appendices are included to provide background on term algebras, fixed point semantics of recursive terms, and natural transformations.

2 M-terms and M-types

We introduce here a language whose expressions denote possibly recursive types involving finite sums and products. We assume given a set $\mathcal{T}_{const}$ of type constants and a set $\mathcal{T}_{var}$ of type variables, disjoint from $\mathcal{T}_{const}$. $\mathcal{T}_{const}$ is assumed at most countable and $\mathcal{T}_{var}$ is assumed countably infinite. In the examples we will assume that $\mathcal{T}_{const}$ contains the constants **int**, **char**, and **1**. The language is defined by means of a ranked alphabet $\mathcal{F}(\mathcal{M})$ whose elements of rank 0 are $\mathcal{T}_{const} \cup \mathcal{T}_{var}$. The operators of positive rank are:

$\times^n$	rank = n	one for each positive integer n
$+^m$	rank = m	one for each positive integer m
in_k	rank = 1	one for each positive integer k
$\mu\ \alpha$	rank = 1	one for each $\alpha \in \mathcal{T}_{var}$

The set of expressions in this language is denoted M-expr ("M-expressions"). M-expressions can be written informally in a linear syntax, as in $\mu\alpha.\mathbf{int} + (\alpha \times \mathbf{char})$, where we have rewritten $+^2$ and $\times^2$ as infix operators. We also think of M-expressions as finite ordered trees [10] labeled by elements of $\mathcal{F}(\mathcal{M})$, in which the number of children of a node must equal the rank of its label. We use freely the fact (see Appendix A.2) that M-expr is the initial $\mathcal{F}(\mathcal{M})$-algebra.

The operators $\times^n$ are used to construct n-ary products, and the sum operators $+^m$ construct m-ary disjoint unions. in_k is the injection of the k-th summand of a disjoint union. The operator $\mu\alpha$ introduces a recursion; it binds free occurrences of α in its body (child) and we have as usual the relation of alpha-equivalence of M-expressions. An alpha-equivalence class is called an M-term. We have substitution in M-terms and so can define unfolding:

Definition 2.1 *If $\tau = \mu\ \alpha.\tau_1$ is a recursion, we define* unfold(τ) *to be* $[\tau/\alpha]\tau_1$.

We will see below that τ and unfold(τ) are semantically equivalent. Note that unfold(τ) is closed if τ is.

2.1 Two Tree Declarations

Here are two declarations (adapted from [5]), in Java and in C, of types representing ordered trees whose nodes each contain an integer and a character. In this section we translate each to an M-term and show informally that the translations are appropriate. In subsequent sections we describe their M-types, derive a coercion between them, and illustrate its computation. The correctness of the translation to M-terms depends on annotations, which are represented here by comments and which we explain below.

The Java class treeJ (Fig. 1) has a "by value" annotation, directing that its methods be ignored, and that we only consider its data structure. (There is also a "by proxy" annotation which would only consider its methods. See also the discussion of port types in Section 9.) Now class valPair should clearly be translated to $\mathbf{int}\times\mathbf{char}$; we abbreviate this M-term as Vp. Let τ and $\tau 1$ be respectively the translations of class treeJ and the array type treeJ[ ]. The forms of these M-terms are motivated by the following expected equivalences, written $\simeq$:

1. $\tau \simeq \mathsf{Vp} \times \tau 1$

2. $\tau 1 \simeq \mathbf{1} + \tau \times \tau 1$

The first equivalence uses the assertion, by annotation, that the two fields "contents" and "children" never contain null references. Null references or pointers are represented by $\mathbf{1}$, whose meaning is a singleton that one may think of as NIL. In the second equivalence we are treating an array of variable length as a list. Thus a value of type treeJ[ ] is represented by $\mathbf{1} \in \mathcal{T}_{const}$, in case the array has length 0, or as a pair (a, b) in which a is a treeJ object and b is another value of type treeJ[ ]. These two related recursive equations lead us to the following M-term:

$$\tau = \mu\alpha_1.((\textbf{int} \times \textbf{char}) \times (\mu\alpha_2.\mathbf{1} + \alpha_1 \times \alpha_2))$$

In the C example (Fig. 2) let σ and σ_1 be the translations of struct treeC and struct treeList respectively. We have the relations:

1. $\sigma \simeq (\mathbf{1} + \sigma_1) \times \textbf{char} \times \textbf{int}$

2. $\sigma_1 \simeq \sigma \times (\mathbf{1} + \sigma_1)$

The possibly null pointers "nextVal" and "children" are represented by sums of the constant $\mathbf{1}$ and the (M-term of) the type pointed to. The resulting M-term is:

$$\sigma = \mu\beta_1.((\mathbf{1} + \mu\beta_2.\beta_1 \times (\mathbf{1} + \beta_2)) \times \textbf{char} \times \textbf{int})$$

We will see in the next section that the values in the M-types denoted by these M-terms are appropriate abstractions of the concrete Java and C objects.

2.2 The model: M-types

The goal of this section is to define sets of values which will be the "meanings" of closed M-terms. We follow the general procedure of Appendix B. Given sets of values for the type constants, the problem is to build up value sets for non-primitive M-terms in a way that reflects the structures of these terms.

So we assume we have sets $[\![A]\!]$, one for each $A \in \mathcal{T}_{const}$, and that these sets are pairwise disjoint. The programming type constants are assumed to have their usual meanings, so for example $[\![\textbf{int}]\!]$ will be the set of signed 32-bit integer values, and $[\![\mathbf{1}]\!]$ is the singleton set $\{1\}$, which we abbreviate as 1. The union of all the sets $[\![A]\!]$, for all $A \in \mathcal{T}_{const}$, is written $\mathcal{PV}$; its elements are called *primitive* values.

```
final class treeJ {         // by value
  ... suitable constructor, accessors, etc. ...
  private valPair contents; // never null
  private treeJ[] children; // never null, but may be empty array
}   // references to treeJ objects are never null
final class valPair {       // by value
  ... suitable constructor, accessors, etc. ...
  private int value;
  private char code;
}
```

Figure 1: Java tree example

```
struct treeC;
struct treeList {
    struct treeC thisVal;
    struct treeList * nextVal; // might be null
};
struct treeC {
    struct treeList * children; // might be null
    char code;
    int value;
};
```

Figure 2: C tree example

The set of *structured values*, written $\mathcal{S}$, is the initial algebra over the ranked alphabet $\mathcal{F}(\mathcal{S})$:

v	rank = 0	one for each v $\in \mathcal{PV}$
$\times^k$	rank = k	one for each positive integer k
in_k	rank = 1	one for each positive integer k

For v $\in \mathcal{S}$, ht(v) is the height of v when viewed as a tree, ie the length of the longest path in dom(v). The elements of height 0 in $\mathcal{S}$ are just the primitive values. For V $\subseteq \mathcal{S}$ and k $\in \mathbb{N}$, we write $\mathrm{V}^{(k)}$ for the set of elements in V of height $\leq$ k.

Definition 2.2 *The set $\mathcal{M}$ is the power set of $\mathcal{S}$ i.e., the set of all subsets of $\mathcal{S}$. Elements of $\mathcal{M}$ are called* M-types.

In the next section we define a mapping which assigns to each (closed) M-term τ a set $[\![\tau]\!] \subseteq \mathcal{S}$, i.e., $[\![\tau]\!] \in \mathcal{M}$. We will consider $[\![\tau]\!]$ to be the set of "values of type τ".

2.3 The semantic function

Like any power set, $\mathcal{M}$ is a pointed cpo. We now give it the structure of a continuous $(\mathcal{F}(\mathcal{M}))'$-algebra (see App. B).

For A in $\mathcal{T}_{const}$ we have $[\![\mathrm{A}]\!] \in \mathcal{M}$, so we can define $\mathrm{A}_{\mathcal{M}} = [\![\mathrm{A}]\!]$. Next we must define, for each non-binding f $\in \mathcal{F}(\mathcal{M})$ of rank q $>$ 0, a continuous function $\mathrm{f}_{\mathcal{M}}$: $\mathcal{M}^q \rightarrow \mathcal{M}$. The cases we have to treat are n-ary sums and products, and the injections in_k.

By construction $\times^n$ acts on $\mathcal{S}$, mapping the n-tuple $\overline{v} = (\mathrm{v}_1,\ldots,\mathrm{v}_n)$ to $\times^n(\mathrm{v}_1,\ldots,\mathrm{v}_n)$. This action extends to $\mathcal{M}$ by mapping an n-tuple $\overline{V} = (\mathrm{V}_1,\ldots,\mathrm{V}_n)$ in $\mathcal{M}^n$ to $\{\times^n(\overline{v})$: $\mathrm{v}_j \in \mathrm{V}_j\} \subseteq \mathcal{S}$. Thus we have defined a function $(\times^n)_{\mathcal{M}}$: $\mathcal{M}^n \rightarrow \mathcal{M}$, which is easily seen to be continuous.

For a positive integer k, the function in_k: $\mathcal{S} \rightarrow \mathcal{S}$ given by (v $\mapsto \mathrm{in}_k$(v)) extends to a function $(\mathrm{in}_k)_{\mathcal{M}}$: $\mathcal{M} \rightarrow \mathcal{M}$ sending V to $\{\mathrm{in}_k(\mathrm{v})$: v $\in$ V$\}$. We then define, for $\overline{V} \in \mathcal{M}^n$

$$(+^n)_{\mathcal{M}}(\overline{V}) = \bigcup_{k=1}^{n}((in_k)_{\mathcal{M}}(V_k))$$

This union is disjoint. It is immediate that the functions $(\mathrm{in}_k)_{\mathcal{M}}$ and $(+^n)_{\mathcal{M}}$ are continuous.

Continuing the procedure of Appendix B we define the cpo of type environments TEnv = $[\mathcal{T}_{var} \rightarrow \mathcal{M}]$, and the semantic domain $\mathcal{D} = [\mathrm{TEnv} \rightarrow \mathcal{M}]$. The semantic function is the unique morphism of $\mathcal{F}(M)$-algebras $[\![\ ]\!]$: M-expr $\rightarrow \mathcal{D}$ satisfying, for any ρ in TEnv

1. for A in $\mathcal{T}_{const}$, $[\![\mathsf{A}]\!].\rho = [\![\mathsf{A}]\!]$. (an abuse of notation, see (1) below)
2. for α in $\mathcal{T}_{var}$, $[\![\alpha]\!].\rho = \rho(\alpha)$
3. for $\overline{\tau} \in (\text{M-expr})^n$, $[\![\times^n(\overline{\tau})]\!].\rho = (\times^n)_{\mathcal{M}}([\![\tau_1]\!].\rho, \ldots, [\![\tau_n]\!].\rho)$
4. for $\tau \in$ M-expr and positive integer k, $[\![\mathsf{in}_k(\tau)]\!].\rho = (\mathsf{in}_k)_{\mathcal{M}}([\![\tau]\!].\rho)$
5. for $\overline{\tau} \in (\text{M-expr})^n$, $[\![+^n(\overline{\tau})]\!].\rho = (+^n)_{\mathcal{M}}([\![\tau_1]\!].\rho, \ldots, [\![\tau_n]\!].\rho)$
6. for α in $\mathcal{T}_{var}$ and $\tau \in$ M-expr, $[\![\mu\ \alpha.\tau]\!].\rho = \mathsf{fix}(\mathsf{W} \mapsto [\![\tau]\!].\rho[\mathsf{x} \mapsto \mathsf{W}])$, where W ranges over $\mathcal{M}$.

All properties of such a semantic mapping (Appendix B.2) apply here. In particular we note the following:

1. If τ is closed, then $[\![\tau]\!]$ is a constant function TEnv$\rightarrow\mathcal{M}$. We abuse notation and also denote this constant value by $[\![\tau]\!]$. (We have already done so in the case of $[\![\mathsf{A}]\!]$ for A $\in \mathcal{T}_{const}$.)
2. $[\![\ ]\!]$ is invariant under alpha-equivalence and hence is defined on M-terms.
3. $[\![\ ]\!]$ is invariant under unfolding: $[\![\tau]\!] = [\![[\tau/\alpha]\tau_1]\!]$ for any M-term $\tau = \mu\alpha.\tau_1$.

2.4 The M-types of treeJ and treeC

Let τ be $\mu\alpha_1.((\mathbf{int} \times \mathbf{char}) \times (\mu\alpha_2.\mathbf{1} + \alpha_1 \times \alpha_2))$, the M-term representing treeJ. Then unfold(τ) = $(\mathbf{int}\times\mathbf{char})\times\ (\mu\alpha_2.\mathbf{1}+\ \tau \times \alpha_2)$. Let's also abbreviate the inner recursion: $\tau_1 = \mu\alpha_2.\mathbf{1}+\ \tau \times \alpha_2$. This M-term represents the list of children, and we have unfold(τ_1) = $\mathbf{1}+\ \tau \times \tau_1$. Since a recursion and its unfolding have the same meaning we find that the expected equivalences are valid:

1. $[\![\tau]\!] = [\![\mathbf{int}\times\mathbf{char}]\!]\ \times[\![\tau_1]\!] = ([\![\mathbf{int}]\!]\times[\![\mathbf{char}]\!]) \times [\![\tau_1]\!]$
2. $[\![\tau_1]\!] = 1 + [\![\tau]\!]\times[\![\tau_1]\!]$

Similarly we consider the M-term representing treeC: $\sigma = \mu\beta_1.((\mathbf{1} + \mu\beta_2.\beta_1 \times (\mathbf{1} + \beta_2)) \times \mathbf{char} \times \mathbf{int})$. We have unfold($\sigma$) = $(\mathbf{1}+\ \mu\beta_2.\sigma\times(\mathbf{1}+\ \beta_2)) \times \mathbf{char}\times\ \mathbf{int}$, and we abbreviate again $\sigma_1 = \mu\beta_2.\sigma\times(\mathbf{1}+\ \beta_2)$. Then unfold($\sigma_1$) = $\sigma\times(\mathbf{1} + \sigma_1)$, so we have the recursive equations:

1. $[\![\sigma]\!] = (1 + [\![\sigma_1]\!]) \times [\![\mathbf{char}]\!] \times [\![\mathbf{int}]\!]$

2. $[\![\sigma_1]\!] = [\![\sigma]\!] \times (1 + [\![\sigma_1]\!])$

Each of the source types describes ordered trees whose nodes contains pairs of integers and characters. Let's show how the values in the two M-types correspond to such trees. We'll look first at τ. From the recursive equations above we see that any $\mathsf{v} \in [\![\tau]\!]$ has the form (vp, ch) with $\mathsf{vp} \in \mathsf{Vp} = [\![\mathbf{int}\times\mathbf{char}]\!]$ and $\mathsf{ch} \in [\![\tau_1]\!]$. vp is the contents of the root of the tree, say (23,"q"), and ch is the list of children. ch has two possible forms: $\mathsf{in}_1(1)$ or $\mathsf{in}_2((\mathsf{v}', \mathsf{ch}')) \in \mathsf{in}_2([\![\tau]\!]\times[\![\tau_1]\!])$. We'll write $\mathsf{in}_1(1) = \mathsf{NIL}_\tau$; it represents an empty list. Here is a list of length 2: $\mathsf{in}_2((\mathsf{v}_1, \mathsf{in}_2((\mathsf{v}_2, \mathsf{NIL}_\tau))))$. Henceforth we will use list notation, so the previous list becomes $(\mathsf{v}_1, \mathsf{v}_2)_\tau$.

We'll now build a tree with 6 nodes $\mathsf{v}_i = (\mathsf{vp}_i, \mathsf{ch}_i)$. v_1 is the root with children v_2 and v_4. v_2 has a single child v_3, and v_4 has two children v_5 and v_6. The contents will be $\mathsf{vp}_1 = (1,\text{"a"}),\ldots, \mathsf{vp}_6 = (6, \text{"f"})$. The leaf nodes are v_3, v_5, and v_6. $\mathsf{v}_3 = ((3, \text{"c"}), \mathsf{NIL}_\tau)$ and the other two have the same shape. Next v_2, with a single child, is $((2,\text{"b"}), (\mathsf{v}_3)_\tau)$, and $\mathsf{v}_4 = ((4,\text{"d"}), (\mathsf{v}_5, \mathsf{v}_6)_\tau)$, and finally $\mathsf{v}_1 = ((1,\text{"a"}), (\mathsf{v}_2, \mathsf{v}_4)_\tau)$.

Values of the other M-term σ have a similar structure except that they are 3-ary products and the constants occur in the reverse order. The equivalent tree has nodes $\mathsf{w}_j \in [\![\sigma]\!]$ for j = 1,..., 6. The M-type $[\![\mathbf{1}+ \sigma_1]\!]$ represents the lists of children with the same structure as $[\![\tau_1]\!]$. We write NIL_σ for $\mathsf{in}_1(1)$ as an element of $[\![\mathbf{1} + \sigma_1]\!]$. So $\mathsf{w}_1 = ((\mathsf{w}_2, \mathsf{w}_4)_\sigma, \text{"a"}, 1)$ is the root, and for a last example $\mathsf{w}_4 = ((\mathsf{w}_5, \mathsf{w}_6)_\sigma, \text{"d"}, 4)$.

We will construct a coercion from τ to σ, in Section 4, and in Section 5 we will observe that this coercion maps v_j to w_j for each j.

3 Polynomial functors and natural transformations

We want to consider algebraic isomorphisms between M-terms. Examples include the associative and commutative laws for sums and products, the distributive law (of products over sums), and the existence of a unit for products. But unfortunately in $\mathcal{M}$ (and also in the model of infinite regular trees) these laws do *not* hold. For instance, $\mathbf{int}\times\mathbf{char}$ denotes a different subset of $\mathcal{S}$ than $\mathbf{char}\times\mathbf{int}$: (6,"w") is in the first but not in the second. On the other hand, there is an obvious 1-1 correspondence ϕ from the first M-type to the second given by $\phi(\mathsf{n}, \mathsf{c}) = (\mathsf{c}, \mathsf{n})$. Thus it is more appropriate to consider algebraic axioms as specifying functions (coercions) rather than identities.

These algebraic coercions will be used not just in isolation but in combination with subordinate coercions. If we have coercions $\mathsf{c}_j\colon \tau_j \to \sigma_j$, then $\phi(\mathsf{c}_1, \mathsf{c}_2)$ is a coercion from $\tau_1 \times \tau_2$ to $\sigma_2 \times \sigma_1$ sending $(\mathsf{v}_1, \mathsf{v}_2)$ to $(\mathsf{c}_2(\mathsf{v}_2), \mathsf{c}_1(\mathsf{v}_1))$. Thus ϕ is a kind of higher-order function, mapping pairs of coercions to a new coercion.

Notice that ϕ really defines a family of functions (V_1, V_2)$\rightarrow$(V_2, V_1), one for each pair $\overline{V}$ = (V_1, V_2) in $\mathcal{M}\times\mathcal{M}$. The source and destination of the function also depend on $\overline{V}$. To express this dependence we write the function as $\phi(\overline{V})$: $t_1(\overline{V})\rightarrow t_2(\overline{V})$. Here t_1 is the "polynomial" term $y_1\times y_2$ and $t_2 = y_2\times y_1$. $t_1(\overline{V})$ is the result of substituting each V_j for each occurrence of y_j in t_1, and similarly for $t_2(\overline{V})$. t_1 and t_2 also define higher-order functions, as follows. If we have functions h_j: $V_j\rightarrow W_j$, we get a function $t_1(\overline{h})$: $t_1(\overline{V})\rightarrow t_1(\overline{W})$, mapping ($v_1$, v_2) to ($h_1(v_1)$, $h_2(v_2)$), and similarly we get $t_2(\overline{h})$: $t_2(\overline{V})\rightarrow t_2(\overline{W})$.

What we have said so far amounts to saying that the t_j define functors and that ϕ is morphism from t_1 to t_2, also known as a natural transformation. For this to make sense, of course, the cpos $\mathcal{M}$ and $\mathcal{M}\times\mathcal{M}$ must also have the structures of categories[2]. In the rest of this section, we define these structures, characterize the class of polynomial functors, and study some properties of natural transformations between them. Natural transformations will be used in the semantic model of coercions (section 5).

3.1 The categories $\mathcal{M}^n$

The elements of $\mathcal{M}$, i.e., the subsets of $\mathcal{S}$, together with all functions between them, form a category. We will abuse notation and also denote this category by $\mathcal{M}$. It has an initial object (the empty set) and a final object (any singleton set). $\times^n(V_1,\ldots,V_n)$ is the product of the V_j in the sense of category theory, where the j-th projection maps $\times^n(\overline{v})$ to v_j. Dually $+^n(\overline{V})$ is a coproduct, with j-th injection mapping v_j to $in_j(v_j)$.

The cpo $\mathcal{M}^n$ can similarly be viewed as the set of objects of a category (also denoted by $\mathcal{M}^n$). An arrow $\overline{f}$: $\overline{V}\rightarrow\overline{W}$ in this category is just an n-tuple of functions f_j: $V_j\rightarrow W_j$, and composition is defined by

$$\overline{g}\circ\overline{f} = (g_1\circ f_1,\ldots,g_n\circ f_n)$$

$\mathcal{M}^n$ is actually a product of categories with projection functors pr_j: $\mathcal{M}^n\rightarrow\mathcal{M}$ given by $pr_j(\overline{V}) = V_j$ and $pr_j(\overline{f}) = f_j$. A functor F: $\mathcal{M}^p\rightarrow\mathcal{M}^q$ is uniquely of the form (F_1, ..., F_q) where $F_j = pr_j\circ F$.

3.2 Polynomial functors

The mapping $\overline{V}\mapsto\times^n(\overline{V})$ is actually the object mapping of a functor $\times^n$: $\mathcal{M}^n\rightarrow\mathcal{M}$. Thus corresponding to each $\overline{f}$: $\overline{V}\rightarrow\overline{W}$, we have a function $\times^n(\overline{f})$ from $\times^n(\overline{V})$ to $\times^n(\overline{W})$. This function simply maps $\times^n(v_1,\ldots,v_n)$ to $\times^n(f_1(v_1),\ldots,f_n(v_n))$.

[2]see App. C and [12] for basic facts about categories

Similarly in_k and $+^n$ define functors. f:V$\to$W induces a map in_k(f): in_k(V)$\to$$\mathsf{in}_k$(W), sending in_k(v) to in_k(f(v)). Then the maps $\mathsf{in}_k(\mathsf{f}_k)$ combine to induce $+^n(\overline{f})$: $+^n(\overline{V})$ to $+^n(\overline{W})$.

We now consider various functors $\mathcal{M}^p \to \mathcal{M}^q$ that can be built up from constants, variables, and sum and product operations.

1. for any A $\in \mathcal{T}_{const}$ and p $\in \mathbb{N}$, there is a constant functor $\mathcal{M}^p \to \mathcal{M}$. which maps any p-tuple $\overline{V}$ to $[\![A]\!]$ and any arrow $\overline{f}$ to the identity function on $[\![A]\!]$.

2. the j-th projection pr_j: $\mathcal{M}^p \to \mathcal{M}$ (j=1, ..., p) maps $\overline{V}$ to V_j and $\overline{f}$ to f_j.

3. $\times^n$: $\mathcal{M}^n \to \mathcal{M}$

4. $+^n$: $\mathcal{M}^n \to \mathcal{M}$

5. given functors F_j: $\mathcal{M}^p \to \mathcal{M}$, for j = 1, ..., q, their product (F_1, ..., F_q): $\mathcal{M}^p \to \mathcal{M}^q$ maps $\overline{V}$ to ($\mathsf{F}_1(\overline{V})$, ..., $\mathsf{F}_q(\overline{V})$), and the j-th component of $\mathsf{F}(\overline{f})$ is just $\mathsf{F}_j(\overline{f})$.

6. given F: $\mathcal{M}^p \to \mathcal{M}^q$ and G: $\mathcal{M}^q \to \mathcal{M}^r$, we have the usual composition of functors G $\circ$ F: $\mathcal{M}^p \to \mathcal{M}^r$.

Definition 3.1 *The class of* polynomial *functors is the smallest set of functors containing those of types 1-4, and closed under products (5) and composition (6).*

Given a property P of morphisms in $\mathcal{M}$, we say an arrow $\overline{f}$ in $\mathcal{M}^n$ has the property if each f_j does. Thus we will speak of inclusions, injections, and so on, and we will say that a functor F: $\mathcal{M}^p \to \mathcal{M}^q$ preserves property P if $\mathsf{F}(\overline{f})$ has property P whenever $\overline{f}$ does.

Theorem 3.1 *A polynomial functor preserves inclusions, injections, surjections, and bijections. Its object map is a continuous function between cpos.*

Proof. Note that saying inclusions are preserved is not the same as saying that injections are preserved. We are asserting that if $\overline{V} \subseteq \overline{W}$ and F is polynomial, then $\mathsf{F}(\overline{V}) \subseteq \mathsf{F}(\overline{W})$. To prove these assertions, one shows, for each property P, that the basic functors, of types 1-4, preserve property P, and that products and compositions of functors preserving P also preserve P. The details are straightforward.

3.3 Polynomial functors arise from finite terms

Consider the alphabet $\mathcal{F}_0$(s) consisting of the n-ary sum and product operators, the type constants $\mathcal{T}_{const}$, and a fresh set of variables $\{y_1, \ldots, y_s\}$. Let Poly_s be the initial algebra over $\mathcal{F}_0$(s). The elements of Poly_s are terms that "look like" polynomials, although no algebraic identities are assumed; this is what led to the name "polynomial" functors. We show here that t $\in$ Poly_s induces a polynomial functor

$$t_{\mathcal{M},s} : \mathcal{M}^s \to \mathcal{M}$$

Its value on an object $\overline{V}$ in $\mathcal{M}^s$ is obtained by substituting V_j for each occurrence of y_j in t, for each j. We now describe its effect on arrows arrows, trivially adapting techniques from [10].

Consider the set K_s of functors from $\mathcal{M}^s$ to $\mathcal{M}$. For the moment, let's fix s and abbreviate K_s by K. We give K the structure of an algebra over $\mathcal{F}_0$(s) as follows. For A $\in$ $\mathcal{T}_{const}$, let A_K be the constant functor defined by $[\![A]\!]$ ($\in \mathcal{M}$). For j = 1,. . . , s, let $(y_j)_K$ be the j-th projection from $\mathcal{M}^s$ to $\mathcal{M}$. Next let f = $\times^n$. We noted above that f defines a functor from $\mathcal{M}^n$ to $\mathcal{M}$. If we have $\phi_1, \ldots, \phi_n \in$ K, each ϕ_j is a functor from $\mathcal{M}^s$ to $\mathcal{M}$, so their product is a functor $\overline{\phi}$: $\mathcal{M}^s \to \mathcal{M}^n$. Then the composition f $\circ$ $\overline{\phi}$ is a functor from $\mathcal{M}^s$ to $\mathcal{M}$, i.e., an element of K. Thus f defines a function $(\times^n)_K$: $K^n \to$ K. The case of $+^n$ is treated similarly.

Hence there is a unique morphism $\mathrm{Poly}_s \to K_s$ of $\mathcal{F}_0$(s)-algebras, and $t_{\mathcal{M},s}$ is the image of t. From now on, when the context is clear, we drop the subscripts and simply write t: $\mathcal{M}^s \to \mathcal{M}$.

Example 3.1 *Take s = 3 and t = $y_1 \times (A + y_3) \in \mathrm{Poly}_3$. Then t($\overline{V}$) = $V_1 \times ([\![A]\!] + V_3)$. Given $\overline{f}$: $\overline{V} \to \overline{W}$ in $\mathcal{M}^3$, we compute*

$$t(\overline{f})(v_1, in_1(a)) = (f_1(v_1), in_1(a)) \textit{ and } t(\overline{f})(v_1, in_2(v_3)) = (f_1(v_1), in_2(f_3(v_3)))$$

with $v_j \in V_j$. Note that f_2 plays no role here.

Since t $\in$ Poly_s defines a functor from $\mathcal{M}^s$ to $\mathcal{M}$, a tuple $(t_1, \ldots, t_r)$ of such terms defines a functor from $\mathcal{M}^s$ to $\mathcal{M}^r$. It is now easy to show

Proposition 3.2 *A functor F: $\mathcal{M}^s \to \mathcal{M}^r$ is polynomial iff it is obtained as above from $\overline{t}$ $\in$ ${\mathrm{Poly}_s}^r$. $\overline{t}$ is uniquely determined by F.*

3.4 Natural Transformations of polynomial functors

Now consider a natural transformation (App. C) ϕ from t_1 to t_2. Given functions $\overline{a}$: $\overline{V}\rightarrow\overline{W}$, there is an induced function $\phi(\overline{a})$: $t_1(\overline{V})\rightarrow t_2(\overline{W})$. $\phi(\overline{a})$ is equal to $\phi(\overline{W}) \circ t_1(\overline{a})$ and also equal to $t_2(\overline{a}) \circ \phi(\overline{V})$.

We have seen that polynomial functors preserve injections. We record the fact that if ϕ and the component morphisms a_j are all injective (resp surjective,...) then so is $\phi(\overline{a})$.

Proposition 3.3 *Let ϕ: $t_1 \rightarrow t_2$ be a morphism between polynomial functors of degree s. Let a_j: $V_j \rightarrow W_j$ be functions in $\mathcal{M}$ (for $j = 1, \ldots, s$). If ϕ is injective (see Def C.4) and each a_j is injective, then $\phi(\overline{a})$: $t_1(\overline{V}) \rightarrow t_2(\overline{W})$ is also injective. Similarly $\phi(\overline{a})$ is surjective (resp bijective) if ϕ and each a_j are surjective (resp. bijective).*

Proof. Immediate from the definition of $\phi(\overline{a})$ as $\phi(\overline{W}) \circ t_1(\overline{a})$.

3.4.1 The set Φ

Our substitute for axioms in what follows will be a set Φ of 4-tuples (ϕ, t_1, t_2, s) where t_1 and $t_2 \in K_s$ and ϕ: $t_1 \rightarrow t_2$ is a natural transformation. Much of what follows does not depend on the particular choice of Φ, which is thus a parameter of the theory. We note here some properties of Φ that are sometimes important.

In examples we will assume Φ contains identity transformations for terms such as $y_1 \times y_2$ or $y_1 + y_2$. In section 7 we require conditions such as injectivity of the functions $\phi(\overline{V})$.

The set of coercions expressible in our language (section 4) depends on the set Φ. For example, if we include only the tuples expressing binary commutativity and associativity of product, then we cannot obtain coercions such as the obvious one from from $\tau\times$**int**$\times$**char** to (**int**$\times\tau$)$\times$**char**, since the first M-term is a ternary product. It is also sometimes convenient to assume that Φ is closed under composition. Thus if we have tuples representing commutativity of sums and the usual form of distributivity of product over sum, then Φ would also contain (ϕ, $y_1\times(y_2 + y_3)$, $y_1\times y_3 + y_1\times y_2$), 3), where ϕ applies the distributive transformation and then commutes the summands.

Here is an example used later (in section 4.2) in deriving a coercion between the M-terms for treeJ and treeC.

Example 3.2 *The tuple is (ϕ_0, $(y_1\times y_2)\times y_3$, $y_3\times y_2\times y_1$, 3). Here ϕ_0 be the obvious natural transformation from t_1 to t_2. So given $\overline{V} \in \mathcal{M}^3$, $\phi_0(\overline{V})$ maps ((v_1, v_2), v_3) to (v_3, v_2, v_1).*

3.5 Extension to partial functions

Let $\mathcal{PF}$ denote the set of all partial functions from $\mathcal{S}$ to itself. Elements of $\mathcal{PF}$ are ordinary (total) functions f: $V \to \mathcal{S}$ where V = dom(f) is a subset of $\mathcal{S}$. We say f is defined at x iff $x \in$ dom(f); the notation $f \downarrow x$ is also used. For subsets W_1 and W_2 of $\mathcal{S}$, we say f maps W_1 into W_2 iff $W_1 \subseteq$ dom(f) and f(W) $\subseteq$ W.

We say, for g and $f \in \mathcal{PF}$, that g extends f iff g is defined wherever f is and has the same value, ie, for any x, $f \downarrow x \Rightarrow (g \downarrow x$ and $g(x) = f(x))$. The relation of extension is a partial order on $\mathcal{PF}$ and we write $f \sqsubseteq g$ iff g extends f. $\mathcal{PF}$ is in fact a cpo, since directedness of a subset D of $\mathcal{PF}$ implies that any two f and g in D agree wherever they are both definied. It is a pointed cpo, with bottom element the empty funtion, and so the fixed point theorem is available.

Partial functions f and g can be composed: $g \circ f$ is defined at x iff f is defined at x and also g is defined at f(x). In this case $(g \circ f)(x) = g(f(x))$.

Proposition 3.4 *Composition is a continuous function $\mathcal{PF} \times \mathcal{PF} \to \mathcal{PF}$.*

One checks this by considering the variables separately, that is, for g fixed, showing that ($f \mapsto g \circ f$) is continuous, and similarly when f is fixed.

Proposition 3.5 *Let t_1 and t_2 be polynomial functors of degree s and let ϕ: $t_1 \to t_2$ be a morphism. Suppose for j = 1, ..., s, we have $u_j \in \mathcal{PF}$.*

1. *$\phi(\overline{u}) = \phi(u_1, \ldots, u_s)$ defines a function whose domain $t_1(dom(u_1), \ldots, dom(u_s))$ and range $t_2(\mathcal{S}, \ldots, \mathcal{S})$ are subsets of $\mathcal{S}$. $\phi(\overline{u})$ is thus an element of $\mathcal{PF}$.*

2. *$(\overline{u} \to \phi(\overline{u}))$ is a continuous function from $\mathcal{PF}^n$ to $\mathcal{PF}$.*

3. *If u_j maps V_j into W_j (j=1, ..., s), then $\phi(\overline{u})$ maps $t_1(\overline{V})$ into $t_2(\overline{W})$.*

Proof. The first assertion is obvious. For the second, ϕ is clearly monotonic in u. One then considers each variable separately and uses continuity (Prop. 3.3). For the third assertion, we have $V_j \subseteq dom(u_j)$ and of course $W_j \subseteq \mathcal{S}$, so let a_j: $V_j \to dom(u_j)$ and b_j:$W_j \to \mathcal{S}$ be the inclusion maps , and let c_j: $V_j \to W_j$ be the restriction of u_j. Then $b_j \circ c_j = u_j \circ a_j$. Now $t_1(\overline{a})$: $t_1(\overline{V}) \to t_1(dom(u_1), \ldots, dom(u_s)))$ is an inclusion, and

$$
\begin{aligned}
\phi(\overline{u}) \circ \mathsf{t}_1(\overline{a}) &= \phi(\overline{c}) \circ \mathsf{t}_1(\overline{u}) \circ \mathsf{t}_1(\overline{a}) \\
&= \phi(\mathsf{u}_1 \circ \mathsf{a}_1, \ldots, \mathsf{u}_s \circ \mathsf{a}_s) \\
&= \phi(\mathsf{b}_1 \circ \mathsf{c}_1, \ldots, \mathsf{b}_s \circ \mathsf{c}_s) \\
&= \mathsf{t}_2(\overline{b} \circ \overline{c}) \circ \phi(\overline{V}) \\
&= \mathsf{t}_2(\overline{b}) \circ \mathsf{t}_2(\overline{c}) \circ \phi(\overline{V}) \\
&= \mathsf{t}_2(\overline{b}) \circ \phi(\overline{c})
\end{aligned}
$$

$\phi(\overline{c})$ maps $\mathsf{t}_1(\overline{V})$ into $\mathsf{t}_2(\overline{W})$, and $\mathsf{t}_2(\overline{b})$ is an inclusion. □

3.6 Further properties

We have shown that $\mathsf{t} \in \mathsf{Poly}_s$ acts as a functor on $\mathcal{M}^s$. It also acts on s-tuples of M-terms $\tau_1, \ldots, \tau_s$ by simultaneous substitution: $\mathsf{t}(\tau_1, \ldots, \tau_s) = [\tau_1/\mathsf{y}_1, \ldots, \tau_s/\mathsf{y}_s]\mathsf{t}$. Note that $\mathsf{t}(\tau_1, \ldots, \tau_s)$ is closed if each τ_j is. The following observation will be needed in Section 4.

Proposition 3.6 *For $t \in Poly_s$ and closed M-terms $\tau_1, \ldots, \tau_s$*

$$[\![t(\tau_1, \ldots, \tau_s)]\!] = t([\![\tau_1]\!], \ldots, [\![\tau_s]\!])$$

Proof. Fix $\overline{\tau}$ and proceed by structural induction on t.

Proposition 3.7 *Let ϕ: $t_1 \to t_2$ be a morphism of polynomial functors of degree s, and suppose given two systems of functions a_j: $V_j \to W_j$ and b_j: $V_j \to W_j$ ($j = 1, \ldots, s$). Suppose for each j that a_j agrees with b_j on a subset U_j, i.e., $a_j(x) = b_j(x)$ for any $x \in U_j$. Then*

$$\phi(\overline{a}) \text{ agrees with } \phi(\overline{b}) \text{ on } t_1(\overline{U})$$

Proof. Let $\overline{d}$: $\overline{U} \to \overline{V}$ be the morphism whose components are the inclusions $\mathsf{U}_j \to \mathsf{V}_j$. By Prop 3.1, $\mathsf{t}_1(\overline{d})$ identifies $\mathsf{t}_1(\overline{U})$ with a subset of $\mathsf{t}_1(\overline{V})$, so we must show $\phi(\overline{a}) \circ \mathsf{t}_1(\overline{d}) = \phi(\overline{b}) \circ \mathsf{t}_1(\overline{d})$.

The hypothesis is that $\overline{a} \circ \overline{d} = \overline{b} \circ \overline{d}$, and so $\mathsf{t}_1(\overline{a}) \circ \mathsf{t}_1(\overline{d}) = \mathsf{t}_1(\overline{b}) \circ \mathsf{t}_1(\overline{d})$. Now $\phi(\overline{a}) \circ \mathsf{t}_1(\overline{d}) = \phi(\overline{W}) \circ \mathsf{t}_1(\overline{a}) \circ \mathsf{t}_1(\overline{d})$, and similarly $\phi(\overline{b}) \circ \mathsf{t}_1(\overline{d}) = \phi(\overline{W}) \circ \mathsf{t}_1(\overline{b}) \circ \mathsf{t}_1(\overline{d})$, so we are done. □

The following property is essential to a later proof that certain recursive functions terminate.

Proposition 3.8 *Consider $t \in Poly_s$ and natural numbers r and k. Suppose no path in t of length less than r ends in a variable y_j (for any $j = 1, \ldots, s$). Then for any $\overline{V} \in \mathcal{M}^s$*

1. $(t(V_1, \ldots, V_s))^{(k+r)} \subseteq t(V_1^{(k)}, \ldots, V_s^{(k)})$

2. *Given* $\overline{W} \in \mathcal{M}^s$ *and* $\overline{f} \in \mathcal{PF}^s$*, suppose each* f_j *maps* $V_j^{(k)}$ *into* W_j*. Then* $t(\overline{f})$ *maps* $(t(\overline{V}))^{(k+r)}$ *into* $t(\overline{W})$

Proof. The case k+r=0 is trivial, so we assume k+r>0. Check (1) by induction on the structure of t. For example if $t = \times^n(t_1, \ldots, t_n)$, each t_j satisfies the hypotheses of (1) for r-1. Then

$$\begin{array}{rcl} (t(\overline{V}))^{(k+r)} & = & (\times^n(t_1(\overline{V}), \ldots, t_n(\overline{V})))^{(k+r)} \\ & \subseteq & \times^n((t_1(\overline{V}))^{(k+r-1)}, \ldots, (t_n(\overline{V}))^{(k+r-1)}) \\ & \subseteq & \times^n(t_1(\overline{V}^{(k)}), \ldots, t_n(\overline{V}^{(k)})) \\ & = & t(V_1^{(k)}, \ldots, V_s^{(k)}) \end{array}$$

In any value of $v = \times^n(\overline{v})$, each v_j has height less than ht(v). This justifies the first $\subseteq$. The second follows by induction, applied to each t_j, and the fact that $\times^n$ preserves inclusions (Prop. 3.1). Assertion (2) follows from (1) and Prop.3.1.

4 The language of coercions

We introduce next a formal language of *coercions* terms in which we can express transformations of M-types. It is in the style of Brandt and Henglein [7]; their language was designed to express coercions from subtypes to supertypes, in a recursive type system permitting function types but not sums or products.

The coercions of interest will be assigned types of the form $(\tau \to \sigma)$, where τ and σ are closed M-terms. As in [7] we will present an inference system in which one can derive judgements of the form $(H \vdash c\colon \tau \to \sigma)$, where H is a *type assignment* specifying the types of free variables in c. The meaning of c (closed) will then be a partial function from $[\![\tau]\!]$ to $[\![\sigma]\!]$.

We build up coercions from two sorts of constants. One is a set CPrim of primitive coercions, of rank 0. We assume CPrim contains all identity maps, ie elements 1_A for each $A \in \mathcal{T}_{const}$. The other set is NT (for "natural transformations"). We assume given a function rank: $NT \to \mathbb{N}$. When we discuss typing, we will assume a bijection $NT \leftrightarrow \Phi$ such that if $f \in NT$ corresponds to $(\phi, t_1, t_2, s) \in \Phi$, then rank(f) = s. We assume CPrim and NT are disjoint. We also assume given a countably infinite set CVar of (coercion) variables.

The set CExpr of coercion expressions is based on a ranked alphabet $\mathcal{F}_{co}$ whose elements of rank 0 are CPrim $\cup$ CVar together with symbols fold_τ and unfold_τ for each closed M-term τ of the form $\mu\ \alpha.\tau_1$. We consider NT as a subset of $\mathcal{F}_{co}$ with rank given by the rank function of NT. For each $\gamma \in$ CVar, "rec γ" is considered as a symbol of rank 1 in $\mathcal{F}_{co}$. Finally the binary composition operator "$\circ$", written in infix form is in $\mathcal{F}_{co}$. Thus the compound forms are

$c_2 \circ c_1$	c_1 followed by c_2
$f(c_1,\ldots,c_s)$	f in NT has rank s; the c_j are coercions
rec γ . c	$\gamma \in$ CVar, c a coercion

Fold and unfold will turn out to denote identity functions; they are needed for type inference. We will informally write n-ary compositions instead of fully parenthesizing, in anticipation of the fact that the meaning of $\circ$ is associative. An expression of the form c = rec γ. c_1 is called *a recursion*. The unary operator "rec γ" binds free occurrences of γ in c; these occurrences represent recursive invocations of τ_1. FV(c) denotes the set of $\gamma \in$ CVar that have free occurrences in c. As usual, renaming of bound variables introduces alpha-equivalence on CExpr.

We say c $\in$ CExpr (or equivalently in CTerm) is *finite* if it contains no recursions. c is then closed and any sub-expression is also finite. We say f $\in$ NT is an identity if it corresponds to $(1_t$, t, t, s$) \in \Phi$ for some t, where 1_t is the identity morphism of the functor t. We say c is *trivial* if it is finite and

1. the only primitive coercions occurring in c are identities 1_A, and

2. any f $\in$ NT occurring in c is an identity.

For example 1_A + unfold$_\sigma$ is trivial. More formally this would be written f(1_A, unfold$_\sigma$), where f corresponds to $(1_t$, t, t, 2$) \in \Phi$ and t = $y_1 + y_2$.

4.1 Typing coercions

The system of typing rules is based on assumptions about the types of primitive coercions and the natural transformations NT. We assume the types of primitive coercions are given by a function Σ(c) = (A$\rightarrow$B), where c $\in$ CPrim and A, B $\in \mathcal{T}_{const}$. We assume the identity maps have the right type: $\Sigma(1_A)$ = (A$\rightarrow$A), for each A $\in \mathcal{T}_{const}$.

As for NT, we assume henceforth a 1-1 correspondence NT $\leftrightarrow \Phi$, mapping f $\in$ NT to $(\phi, t_1, t_2, s) \in \Phi$ such that rank(f)= s. We anticipate now our semantic discussion in order to motivate the typing rule for terms of the form c = f(c_1,..., c_s). ϕ will be used to combine the effects of coercions $c_j{:}\tau_j \to \sigma_j$. In our semantic model each c_j defines a partial function from $[\![\tau_j]\!]$ to $[\![\sigma_j]\!]$. Then we have (see Prop. 3.5) a partial function

$$\phi([\![c_1]\!], \ldots, [\![c_s]\!]) : t_1([\![\tau_1]\!], \ldots, [\![\tau_s]\!]) \to t_2([\![\sigma_1]\!], \ldots, [\![\sigma_s]\!])$$

If we set $\tau = \mathrm{t}_1(\tau_1, \ldots, \tau_s)$ and $\sigma = \mathrm{t}_2(\sigma_1, \ldots, \sigma_s)$, then Prop. 3.6 gives

$$[\![\tau]\!] = [\![t_1(\tau_1, \ldots, \tau_s)]\!] = t_1([\![\tau_1]\!], \ldots, [\![\tau_s]\!])$$

and

$$[\![\sigma]\!] = [\![t_2(\sigma_1, \ldots, \sigma_s)]\!] = t_2([\![\sigma_1]\!], \ldots, [\![\sigma_s]\!])$$

We would therefore like to give $\mathrm{f}(\mathrm{c}_1, \ldots, \mathrm{c}_s)$ the type $(\tau \to \sigma)$.

The typing rules (Fig. 3) allow one to derive sequents of the form ($\mathrm{H} \vdash \mathrm{c}{:}\ \tau \to \sigma$), in which c is a coercion term, τ and σ are closed M-terms. H is a *type assignment*, which is a finite list of "hypotheses" of the form $(\gamma{:}\tau \to \sigma)$, where $\gamma \in \mathrm{CVar}$, and τ and σ are *closed* M-terms; a particular γ may not occur more than once. Type assignments may be written as concatenations, as in (H_1, $\gamma{:}\tau_1 \to \tau_2$, H_2). One can show [12] that the elements of H may be reordered without changing the validity of a sequent, so we can also think of H as a function from a finite set $\mathrm{dom(H)} \subseteq \mathrm{CVar}$. In this spirit we write $\mathrm{H}(\gamma) = (\omega_1 \to \omega_2)$, meaning that ($\gamma$: $\omega_1 \to \omega_2$) occurs in H. If H is empty, the sequent is written ($\vdash$ c: $\tau \to \sigma$), and we say c has type $(\tau \to \sigma)$.

(The following discussion may be skipped by readers who already gloss over the issues of renaming and alpha-equivalence. There is an implicit side-condition in rule (**Rec**). If you read the rule as applying to "raw" expressions (as opposed to alpha-equivalence classes of such expressions), you see that γ must not occur in H. Otherwise "H, $\gamma : \tau \to \sigma$" would be ill-formed. So in general we should restrict ourselves to sequents ($\mathrm{H} \vdash c : \tau \to \sigma$) such that no $\gamma \in \mathrm{CVar}$ occurs both in H and as a bound variable in c. Under this assumption, if c′ is alpha-equivalent to c, ($\mathrm{H} \vdash c : \tau \to \sigma$) can be derived if and only if ($\mathrm{H} \vdash c' : \tau \to \sigma$) can be derived. So one may safely read the "c" in a sequent as a coercion term, ie an alpha-equivalence class.)

Henceforth when we write a sequent ($\mathrm{H} \vdash$ c: $\tau \to \sigma$), there is an implicit assumption that it is derivable from our rules. We then say that (H, c) has type $(\tau \to \sigma)$, or that c has a type relative to H. It turns out [12] that in this case $\mathrm{FV(c)} \subseteq \mathrm{dom(H)}$ and that the type only depends on the types assigned to $\gamma \in \mathrm{FV(c)}$.

4.2 Finding a coercion from treeJ to treeC

The typing rules characterize proofs of typing judgements but of course can also be used to guide a search for a coercion c of a given type. We illustrate this with the M-terms used in section 2.1:

$$\text{for c} \in \text{CPrim}, (H \vdash c : \Sigma(c)) \qquad \textbf{(Cprim)}$$

$$(H, \gamma : \tau \to \sigma, H' \vdash \gamma : \tau \to \sigma) \qquad \textbf{(Hyp)}$$

$$(H \vdash fold_{\mu\alpha.\tau} : [\mu\alpha.\tau/\alpha]\tau \to \mu\alpha.\tau) \qquad \textbf{(Fold)}$$

$$(H \vdash unfold_{\mu\alpha.\tau} : \mu\alpha.\tau \to [\mu\alpha.\tau/\alpha]\tau) \qquad \textbf{(Unfold)}$$

$$\frac{(H \vdash c_1 : \tau_1 \to \tau_2) \quad (H \vdash c_2 : \tau_2 \to \tau_3)}{(H \vdash c_2 \circ c_1 : \tau_1 \to \tau_3)} \qquad \textbf{(Comp)}$$

$$\frac{(H \vdash c_j : \tau_j \to \sigma_j) \quad (f \in NT \leftrightarrow (\phi, t_1, t_2, s) \in \Phi)}{(H \vdash f(c_1, \ldots, c_p) : t_1(\overline{\tau}) \to t_2(\overline{\sigma}))} \qquad \textbf{(Nat)}$$

$$\frac{(H, \gamma : \tau \to \sigma \vdash c : \tau \to \sigma)}{(H \vdash \text{ rec}\, \gamma.\ c : \tau \to \sigma)} \qquad \textbf{(Rec)}$$

Figure 3: Typing Rules for Coercions

$$\tau = \mu\alpha_1.((\textbf{int}\times\textbf{char})\times (\mu\alpha_2.\mathbf{1} + \alpha_1 \times \alpha_2)) \qquad \mathsf{treeJ}$$

$$\sigma = \mu\beta_1.((\mathbf{1} + \mu\beta_2.\beta_1\times(\mathbf{1} + \beta_2)) \times \textbf{char} \times \textbf{int}) \qquad \mathsf{treeC}$$

We seek a coercion c satisfying ($\vdash$ c:$\tau \to \sigma$). In this derivation, we will always choose first to unfold recursions, so we look for c_1 such that ($\vdash c_1$: unfold(τ) $\to$ unfold(σ)). Then by the fold and unfold axioms and the composition rule, c = $\text{fold}_\sigma \circ c_1 \circ \text{unfold}_\tau$ will work. Now

$$\text{unfold}(\tau) = (\textbf{int} \times \textbf{char}) \times (\mu\alpha_2.\mathbf{1} + \tau \times \alpha_2)$$

$$\text{unfold}(\sigma) = (\mathbf{1} + \mu\beta_2.\sigma \times (\mathbf{1} + \beta_2)) \times \textbf{char}\times \textbf{int}$$

We recognize these two M-terms as having the forms $(y_1 \times y_2) \times y_3$ and $y_3 \times y_2 \times y_1$, which suggests a coercion $f_0(id_{\textbf{int}}, id_{\textbf{char}}, c_2)$, where $f_0 \in$ NT names the tuple described in Ex. 3.2. c_2 must then be a coercion from $(\mu\alpha_2.\mathbf{1}+ \tau \times \alpha_2)$ to $(\mathbf{1} + \mu\beta_2.\sigma \times (\mathbf{1} + \beta_2))$. Let's abbreviate:

$$\tau_1 = \mu\alpha_2.\mathbf{1} + \tau\times\alpha_2$$

$$\sigma_1 = \mu\beta_2.\sigma \times (\mathbf{1} + \beta_2)$$

$$\sigma_2 = \mathbf{1} + \sigma_1$$

Since τ_1 and σ_1 involve the original M-terms τ and σ, one might anticipate being led back to a coercion from τ to σ, and then as above to a coercion between their unfoldings. This suggests wrapping f_0 in a recursion, so we take

$$c_1 = \text{rec } \gamma_1.\ f_0(1_{\mathbf{int}}, 1_{\mathbf{char}}, c_2)$$

We want ($\vdash c_1$: unfold(τ)$\rightarrow$unfold(σ)), so by (**Rec**) we need

$$(H_1 \vdash f_0(1_{\mathbf{int}}, 1_{\mathbf{char}}, c_2)\text{: unfold}(\tau)\rightarrow\text{unfold}(\sigma))$$

where $H_1 = (\gamma_1\text{: unfold}(\tau)\rightarrow\text{unfold}(\sigma))$. By (**Nat**) we need

$$(H_1 \vdash 1_{\mathbf{int}}\text{: } \mathbf{int}\rightarrow\mathbf{int})$$

$$(H_1 \vdash 1_{\mathbf{char}}\text{: } \mathbf{char}\rightarrow\mathbf{char})$$

$$(H_1 \vdash c_2\text{: } \tau_1 \rightarrow \mathbf{1} + \sigma_1)$$

The first two judgements are instances of (**Cprim**), so we now pursue c_2. unfold(τ_1) = $\mathbf{1} + \tau \times \tau_1$, so set $c_2 = c_3 \circ \text{unfold}_{\tau_1}$ and try for ($H_1 \vdash c_3$: $\mathbf{1} + \tau \times \tau_1 \rightarrow \mathbf{1} + \sigma_1$). We will take $c_3 = \text{rec } \gamma_2.1_{\mathbf{1}} + c_4$. Recall that $1_{\mathbf{1}} + c_4$ is more formally written as $f_+(1_{\mathbf{1}}\ c_4)$, where $f_+ \in$ NT names the identity transformation of $y_1 + y_2$.

In general we may always wrap a natural transformation, such as f_+ above, in a recursion. This is unnecessary when the coercion turns out not to be used recursively, as will be the case for c_5 below. No harm is done because we will see that rec $\gamma.c'$ and c' have the same meaning whenever $\gamma \notin FV(c')$. We can also backtrack in the derivation (we will do this with c_5) and omit the recursion once we see that its subcoercions don't require it.

We now need ($H_2 \vdash c_4$: $\tau \times \tau_1 \rightarrow \sigma_1$) where $H_2 = (H_1, \gamma_2\text{: } \mathbf{1} + \tau \times \tau_1 \rightarrow \mathbf{1} + \sigma_1)$. Unfold($\sigma_1$) = $\sigma\times(\mathbf{1} + \sigma_1)$, so let $c_4 = \text{fold}_{\sigma_1} \circ c_5$, and then $c_5 = \text{rec } \gamma_3.c_6\times c_7$. We now set $H_3 = (H_2, \gamma_3\text{: } \tau \times \tau_1 \rightarrow \sigma \times (\mathbf{1} + \sigma_1))$. We must have ($H_3 \vdash c_6$: $\tau \rightarrow \sigma$) and ($H_3 \vdash c_7$: $\tau_1 \rightarrow \mathbf{1} + \sigma_1$). The reader should have the feeling of déja vu: once again we need a coercion from τ to σ. If we had previously introduced a recursion at this point, we would be done. Instead we follow the previous path and take $c_6 = \text{fold}_\sigma \circ c_8 \circ \text{unfold}_\tau$, looking for c_8 satisfying ($H_3 \vdash c_8$: unfold(τ)$\rightarrow$unfold(σ)). Now we can terminate the recursion, by use of (**Hyp**), and take $c_8 = \gamma_1$.

It remains to take $c_7 = c_9 \circ \text{unfold}_{\tau_1}$, with ($H_3 \vdash c_9$: $\mathbf{1} + \tau \times \tau_1 \rightarrow \mathbf{1} + \sigma_1$). Again using (Hyp), set $c_9 = \gamma_2$ and we *are* done. There are no more unknown c_j's, but there is one possible simplification, mentioned above. If we substitute back into c_5, we get

$$c_5 = \text{rec } \gamma_3.\ (\text{fold}_\sigma \circ \gamma_1 \circ \text{unfold}_\tau) \times (\gamma_2 \circ \text{unfold}_{\tau_1})$$

γ_3 has no free occurrences in the body of the recursion, so we take instead

$$c_5 = (\text{fold}_\sigma \circ \gamma_1 \circ \text{unfold}_\tau) \times (\gamma_2 \circ \text{unfold}_{\tau_1})$$

We recall here the rest of the steps:

$$c_2 = (\text{rec } \gamma_2.\ 1_{\mathbf{1}} + \text{fold}_{\sigma_1} \circ c_5) \circ \text{unfold}_{\tau_1}$$

$$c = \text{fold}_\sigma \circ (\text{rec } \gamma_1.\ f_0(1_{\mathbf{int}}, 1_{\mathbf{char}}, c_2)) \circ \text{unfold}_\tau$$

4.3 Finding coercions in general

One may ask for a procedure which given a pair (τ, σ) of closed M-terms finds a coercion term c such that $(\vdash c\colon \tau \to \sigma)$ if one exists, and otherwise fails. We do not have such a procedure and suspect that one does not exist. This contrasts with the situation for recursive subtyping reported in [7]. There Brandt-Henglein axiomatized the subtype relation $\leq$ for the language of function types with recursion. They also gave a decision procedure (Algorithm S) which would return a proof of $(\tau \leq \sigma)$, if one exists, and otherwise would report failure.

For the purposes of the Mockingbird tool, we have considered an adaptation of Algorithm S to our setting. It guided the strategy followed in the previous section. We replace the use of rule Arrow/Fix with a non-deterministic application of a natural transformation. Our algorithm $\mathcal{C}$ takes as input a triple (H, σ, τ), where H is a type assignment, and searches for a coercion c such that $(H \vdash c{:}\tau \to \sigma)$. $\mathcal{C}(H, \tau, \sigma)$ is computed as follows:

1. if τ is a recursion, return $\mathcal{C}(H, \text{unfold}(\tau), \sigma) \circ \text{unfold}_\tau$
2. if σ is a recursion, return $\text{fold}_\sigma \circ \mathcal{C}(H, \tau, \text{unfold}(\sigma))$
3. if $(\gamma\colon \tau \to \sigma)$ appears in H, return γ
4. if there exists $f \in NT$ corresponding to $(\phi, t_1, t_2, s) \in \Phi$, and there exist M-terms $\tau_1, \ldots, \tau_s$ and $\sigma_1, \ldots, \sigma_s$ such that we have $\tau = t_1(\overline{\tau})$ and $\sigma = t_2(\overline{\sigma})$ then let $H' = (H, \gamma{:}\tau \to \sigma)$ and return rec $\gamma.f(\overline{c})$, where $c_j = \mathcal{C}(H', \tau_j, \sigma_j)$
5. if there exists $c \in \text{CPrim}$ with $\Sigma(c) = (\tau \to \sigma)$ return c
6. else fail

The algorithm is non-deterministic because there may be more than one (f, $\overline{\tau}$, $\overline{\sigma}$) matching (τ, σ). Its power depends (see section 3.4.1) on the choice of Φ. Termination is not obvious.

From a practical point of view, the situation seems more encouraging. In the Mockingbird setting the search for a coercion is undertaken only if the programmer believes the two surface types are equivalent. Since the complexity of such examples is limited by what the programmer can grasp, a bounded search (of the space of all derivations) may often succeed. This approach also makes moot the question of whether the decision algorithm terminates.

5 The semantics of coercions

Our model of closed typed coercions is the set $\mathcal{PF}$ of all partial functions from $\mathcal{S}$ to itself. To obtain a semantic mapping we again apply Theorem B.4, which requires specifying, for each non-binding c $\in \mathcal{F}_{co}$ of rank n, a continuous map $c_{\mathcal{PF}}$: $\mathcal{PF}^n \to \mathcal{PF}$.

We assume that for c $\in$ CPrim we are given in advance functions $c_{\mathcal{PF}}$: $[\![A]\!] \to [\![B]\!]$, where $\Sigma(c) = (A \to B)$, and we assume that for each A $\in \mathcal{T}_{const}$, $[\![1_A]\!] = 1_{[\![A]\!]}$, the identity map of $[\![A]\!]$. For c = fold_τ or c = unfold_τ, we set $c_{\mathcal{PF}} = 1_{[\![\tau]\!]}$, where $\tau = \mu\alpha.\tau_1$ is a closed M-term. This makes sense in view of the typing rules and the fact (see prop. B.10) that $[\![\tau]\!] = [\![[\tau/\alpha]\tau_1]\!]$. Thus for each c $\in \mathcal{F}_{co}$ of rank 0, we have defined $c_{\mathcal{PF}} \in \mathcal{PF}$.

We have seen (Prop. 3.4) that composition of partial functions yields a continuous function $\mathcal{PF}\times\mathcal{PF}\to\mathcal{PF}$. It remains to define continuous functions $f_{\mathcal{PF}}$:$\mathcal{PF}^s\to\mathcal{PF}$ for each f $\in$ NT corresponding to (ϕ, t_1, t_2, s) $\in \Phi$. This was shown in Prop.3.5. We have thus given $\mathcal{PF}$ the structure of continuous algebra over $(\mathcal{F}_{co})'$.

So we introduce the cpo of *coercion* environments CEnv = [CVar $\to \mathcal{PF}$], where CVar is viewed as a discrete cpo. Since CVar is discrete, CEnv is the set of all functions from CVar to $\mathcal{PF}$. The target of the semantic map will be $\mathcal{E}$ = [CEnv $\to \mathcal{PF}$]. By Theorem B.4 there is a unique morphism $[\![\ ]\!]$: CExpr $\to \mathcal{E}$ satisfying, for any ρ in CEnv,

1. for c in CPrim, $[\![c]\!].\rho = c_{\mathcal{PF}}$

2. for γ in Cvar, $[\![\gamma]\!].\rho = \rho(\gamma)$

3. for c_1 and c_2 in CExpr, $[\![c_2 \circ c_1]\!].\rho = ([\![c_2]\!].\rho) \circ ([\![c_1]\!].\rho)$

4. for f $\in$ NT of rank s, corresponding to (ϕ, t_1, t_2, s) $\in \Phi$, and $c_1, \ldots, c_s \in$ CExpr, $[\![f(c_1, \ldots, c_s)]\!].\rho = \phi([\![c_1]\!].\rho, \ldots, [\![c_s]\!].\rho)$

5. for $\gamma \in$ Cvar, c $\in$ CExpr, $[\![\mathsf{rec}\ \gamma.c]\!].\rho = \mathsf{fix}(\mathsf{u} \mapsto [\![\mathsf{c}]\!].\rho[\mathsf{x} \mapsto \mathsf{u}])$, where u ranges over $\mathcal{PF}$.

We record for later reference this immediate consequence:

Proposition 5.1 *Let $g = [\![rec\ \gamma.c]\!].\rho$, with c a coercion term and $\rho \in$ CEnv. Set $\rho' = \rho[\gamma \mapsto g]$. Then $g = [\![c]\!].\rho'$.*

All other general properties of such semantic mappings apply here (see App. B.2). In addition we will need the following fact about trivial coercions.

Proposition 5.2 *If $c \in$ CExpr is finite and ($H \vdash c\colon \tau \to \sigma$), then for any $\rho \in$ CEnv*

1. *$[\![c]\!].\rho$ maps $[\![\tau]\!]$ into $[\![\sigma]\!]$.*
2. *if c is trivial, then $[\![\tau]\!] = [\![\sigma]\!]$ and $[\![c]\!].\rho = 1_{[\![\tau]\!]}$*

The proof is a straightforward induction on the structure of c.

5.1 treeJ and treeC revisited

We will evaluate the coercion term c computed in section 4.2. Let $\mathsf{h} = [\![c]\!].\rho$, where $\rho \in$ CEnv is arbitrary. Since c is closed, h is independent of ρ, and we expect h to map $[\![\tau]\!]$ into $[\![\sigma]\!]$.

We have $\mathsf{h} = ([\![\mathsf{fold}_\sigma]\!].\rho) \circ ([\![\mathsf{c}_1]\!].\rho) \circ ([\![\mathsf{unfold}_\tau]\!].\rho) = [\![\mathsf{c}_1]\!].\rho$, since folds and unfolds denote identity maps. By Prop. 5.1 $[\![\mathsf{c}_1]\!].\rho = [\![\mathsf{f}_0(1_{\mathbf{int}}, 1_{\mathbf{char}}, \mathsf{c}_2)]\!].\rho_1 = \phi_0(1_{[\![\mathbf{int}]\!]}, 1_{[\![\mathbf{char}]\!]}, [\![\mathsf{c}_2]\!].\rho_1)$ where $\rho_1 = \rho[\gamma_1 \mapsto \mathsf{h}]$.

Set $\mathsf{h}_1 = [\![\mathsf{c}_2]\!].\rho_1 = [\![\mathsf{c}_3]\!].\rho_1$, again because an unfolding denotes an identity map. Since c_3 is a recursion, Prop. 5.1 gives $\mathsf{h}_1 = [\![1_{\mathbf{1}} + \mathsf{c}_4]\!].\rho_2$, where $\rho_2 = \rho_1[\gamma_2 \mapsto \mathsf{h}_1]$. $[\![\mathsf{c}_4]\!].\rho_2 = [\![\mathsf{c}_5]\!].\rho_2$. Recalling that

$$\mathsf{c}_5 = (\mathsf{fold}_\sigma \circ \gamma_1 \circ \mathsf{unfold}_\tau) \times (\gamma_2 \circ \mathsf{unfold}_{\tau_1})$$

we get $[\![\mathsf{c}_5]\!].\rho_2 = ([\![\gamma_1]\!].\rho_2) \times ([\![\gamma_2]\!].\rho_2) = \rho_1(\gamma_1) \times \rho_2(\gamma_2) = \mathsf{h} \times \mathsf{h}_1$. We summarize:

$$\mathsf{h} = \phi_0(1_{[\![\mathbf{int}]\!]}, 1_{[\![\mathbf{char}]\!]}, \mathsf{h}_1)\colon ([\![\mathbf{int}]\!] \times [\![\mathbf{char}]\!]) \times [\![\tau_1]\!] \to (1 + [\![\sigma_1]\!]) \times [\![\mathbf{char}]\!] \times [\![\mathbf{int}]\!]$$

$$\mathsf{h}_1 = 1 + \mathsf{h} \times \mathsf{h}_1\colon 1 + [\![\tau]\!] \times [\![\tau_1]\!] \to 1 + [\![\sigma]\!] \times (1 + [\![\sigma_1]\!])$$

The notation above suggests something we don't know yet, that h and h_1 map the indicated sources into the indicated destinations. We will show this is the case in the next section, and in section 7 will show that h and h_1 are isomorphisms.

We conclude by showing that h maps the sample (treeJ) value v_6 to the (treeC) value w_6 (see Section 2.4). First consider the mapping of "children" lists given by h_1, and show that $h_1(\mathsf{NIL}_\tau) = \mathsf{NIL}_\sigma$, and for any tuple $\overline{v'} = (v'_1, \ldots,$ we have $v'_n)$, $h_1((\overline{v'})_\tau) = (\overline{w'})_\sigma$, where $w'_j = h_1(v'_j)$. In fact $h_1(\mathsf{NIL}_\tau) = h_1(in_1(1)) = in_1(1) = \mathsf{NIL}_\sigma$. And for $n > 0$, $h_1((\overline{v'})_\tau) = h_1(in_2(v'_1, (v'_2, \ldots, v'_n)_\tau)) = in_2((h(v'_1), h_1((v'_2, \ldots, v'_n)_\tau)))$. By induction, this is $in_2((w'_1, (w'_2, \ldots, w'_n)_\sigma)), = (\overline{w'})_\sigma$.

Now for a value $((n, c), ch) \in [\![\tau]\!] \times [\![\tau_1]\!] \subseteq [\![\tau]\!]$, we have $h(((n, c), ch)) = (h_1(ch), c, n)$, We consider first the leaf node $v_3 = ((3, \text{"c"}), \mathsf{NIL}_\tau)$. $h(v_3) = h(((3, \text{"c"}), \mathsf{NIL}_\tau)) = (h_1(\mathsf{NIL}_\tau), \text{"c"}, 3) = (\mathsf{NIL}_\sigma, \text{"c"}, 3) = w_3$. Similarly h maps v_5 to v_6 and v_6 to w_6.

One continues by induction on the height of v_j. For example consider the root. $h(v_1) = h(((1, \text{"a"}), (v_2, v_4)_\tau) = (h_1((v_2, v_4)_\tau), \text{"a"}, 1)$, which by induction is equal to $((w_2, w_4)_\sigma, \text{"a"}, 1) = w_1$.

6 Termination

For general coercions, the recursions in $[\![c]\!].\rho$ may diverge, and so the partial function may be undefined at some values in $[\![\tau]\!]$. In this section we introduce a class of coercions for which termination can be proved. Using the information provided by typing, we want to show that a (closed) c which has type $(\tau \to \sigma)$ actually defines a total function from $[\![\tau]\!]$ to $[\![\sigma]\!]$.

Without further restrictions, this is certainly false. For example consider $c = \mathrm{rec}\ \gamma.\gamma$. For any τ and σ, set $H = (\gamma{:}\tau \to \sigma)$. Then $(H \vdash \gamma{:}\ \tau \to \sigma)$, so by (Rec) we get $(\ \vdash c{:}\tau \to \sigma)$. But in fact $[\![c]\!].\rho$ is the bottom element of $\mathcal{PF}$ diverging everywhere. To see this, note that $[\![c]\!].\rho$ is defined as $\mathrm{fix}(g \mapsto [\![\gamma]\!].\rho[\gamma \mapsto g]) = \mathrm{fix}(g \mapsto g) = \perp_{\mathcal{PF}}$.

To achieve the desired convergence (Theorem 6.2) we impose a condition on coercions.

Definition 6.1 *A* constrained *coercion c is one satisfying the following:*

1. *for any subterm of c of the form $c_2 \circ c_1$, either c is closed, or c_1 or c_2 is trivial*
2. *the body of any recursion in c is a natural transformation, ie, for any subterm of c of the form rec $\gamma.c'$, c' has the form $f(c_1, \ldots, c_s)$ where $f \in NT$ and $s > 0$. Let $f \leftrightarrow (\phi, t_1, t_2, s) \in \Phi$. Then we also assume that both t_1 and t_2 are of height > 0.*

These conditions will be used to guarantee that recursive invocations of a natural transformation make progress. Any subterm of a constrained coercion is constrained, and any finite coercion is constrained. Coercions discovered by the Mockingbird tool are in fact constrained.

In the proof of Theorem 6.2, we will need to consider open coercions c whose meanings do depend on the coercion environment. We next define conditions on ρ which guarantee good behavior of $[\![c]\!].\rho$.

Definition 6.2 *Let H be a coercion type assignment and c a coercion such that ($H \vdash c\colon \tau \to \sigma$). We say $\rho \in$ CEnv is compatible with (H, c) iff for any $\gamma \in$ CVar and closed M-terms ω_1 and ω_2*

$$\textit{if } \gamma \textit{ is free in c and } H(\gamma) = (\omega_1 \to \omega_2) \textit{ then } \rho.\gamma \textit{ maps } [\![\omega_1]\!] \textit{ into } [\![\omega_2]\!].$$

We write CEnv(H, c) $= \{\rho \in$ *CEnv: ρ is compatible with (H, c)*$\}$. *For k $\in \mathbb{N}$, we define* CEnv(H, c, k) *as the subset of CEnv consisting of those ρ satisfying the weaker condition*

$$\textit{if } \gamma \textit{ is free in c and } H(\gamma) = (\omega_1 \to \omega_2) \textit{ then } \rho(\gamma) \textit{ maps } [\![\omega_1]\!]^{(k)} \textit{ into } [\![\omega_2]\!].$$

The following facts follow at once from the definitions.

Proposition 6.1 *Suppose ($H \vdash c\colon \tau \to \sigma$). Then*

1. *if c is closed, CEnv(H, c) = CEnv*
2. *for k $\in$ Nat, CEnv(H, c) $\subseteq$ CEnv(H, c, k+1) $\subseteq$ CEnv(H, c, k) $\subseteq$ CEnv*
3. CEnv(H, c) $= \bigcap_{k \in Nat}$ CEnv(H, c, k)
4. *If also ($H \vdash c'\colon \tau' \to \sigma'$), and if FV(c) $\subseteq$ FV(c'), then for any k,*

$$CEnv(H, c', k) \subseteq CEnv(H, c, k)$$

Theorem 6.2 *Let c be a* constrained *coercion, H a coercion type assignment, τ and σ closed M-terms such that ($H \vdash c\colon \tau \to \sigma$). Then for $\rho \in$ CEnv(H, c), $[\![c]\!].\rho$ maps $[\![\tau]\!]$ into $[\![\sigma]\!]$.*

Corollary 6.3 *Let c be a closed and constrained coercion and H a type assignment such that ($H \vdash c\colon \tau \to \sigma$). Then $[\![c]\!]$ maps $[\![\tau]\!]$ into $[\![\sigma]\!]$.*

Proof. (of theorem) We will actually prove the stronger assertion that, for any $k \in \mathbb{N}$ and $\rho \in \mathrm{CEnv}(H, c, k)$, $[\![c]\!].\rho$ maps $[\![\tau]\!]^{(k)}$ into $[\![\sigma]\!]$. This analysis by levels is used to permit an induction on k in the case that c is a recursion. The proof proceeds by induction on the structure of c. If $c \in \mathrm{CPrim} \cup \mathrm{CVar}$, or if c is a fold or unfold, the argument is trivial.

We look closely at the argument in the case $c = c_2 \circ c_1$, in order to explain the purpose of the restriction on compositions. We have $[\![c]\!].\rho = ([\![c_2]\!].\rho) \circ ([\![c_1]\!].\rho)$. There is some closed M-term ω such that $(H \vdash c_1\colon \tau \to \omega)$ and $(H \vdash c_2\colon \omega \to \sigma)$. Since $\mathrm{FV}(c_j) \subseteq \mathrm{FV}(c)$, we have $\rho \in \mathrm{CEnv}(H, c_j, k)$ for $j = 1, 2$. By induction $[\![c_1]\!].\rho$ maps $[\![\tau]\!]$ into $[\![\omega]\!]$, and $[\![c_2]\!].\rho$ maps $[\![\omega]\!]^k$ into $[\![\sigma]\!]$. The difficulty is that we do not know that $[\![c_1]\!].\rho$ maps $[\![\tau]\!]$ into $[\![\omega]\!]^k$. Now use the assumption that c is constrained. If c is closed, so is each of c_1 and c_2, so by induction $[\![c_1]\!].\rho$ maps $[\![\tau]\!]$ into $[\![\omega]\!]$ and $[\![c_2]\!].\rho$ maps $[\![\omega]\!]$ into $[\![\sigma]\!]$, so in fact $[\![c]\!].\rho$ maps $[\![\tau]\!]$ into $[\![\sigma]\!]$. And if c is not closed, then one of c_1 or c_2 is trivial, so the corresponding $[\![c]\!]$ is an identity map and the result is clear.

Next consider the case $c = f(c_1,\ldots,c_s)$ with $f \in \mathrm{NT}$ and $f \leftrightarrow (\phi, t_1, t_2, s) \in \Phi$. Consider $\rho \in \mathrm{CEnv}(H, c, k)$. We must show $[\![c]\!]$ maps $[\![\tau]\!]^{(k)}$ into $[\![\sigma]\!]$. There exist closed M-terms τ_j and σ_j for $j = 1,\ldots, s$ such that $(H \vdash c_j\colon \tau_j \to \sigma_j)$, $\tau = t_1(\tau_1,\ldots,\tau_s)$ and $\sigma = t_2(\sigma_1,\ldots,\sigma_s)$. We have $[\![\tau]\!] = t_1([\![\tau_1]\!],\ldots, [\![\tau_s]\!])$ and $[\![\sigma]\!] = t_2([\![\sigma_1]\!],\ldots, [\![\sigma_s]\!])$. Now from $\mathrm{FV}(c_j) \subseteq \mathrm{FV}(c)$ it follows that $\mathrm{CEnv}(H, c, k) \subseteq \mathrm{CEnv}(H, c_j, k)$. By induction $[\![c_j]\!].\rho$ maps $[\![\tau_j]\!]^{(k)}$ into $[\![\sigma_j]\!]$. Proposition 3.8 implies that $\phi([\![c_1]\!].\rho,\ldots, [\![c_s]\!].\rho)$ maps $[\![\tau]\!]^{(k)}$ into $[\![\sigma]\!]$.

In the final case $c = \mathrm{rec}\ \gamma.\ c'$. We may assume, by renaming, that $\gamma \notin \mathrm{dom}(H)$. We have $(H' \vdash c'\colon \tau \to \sigma)$, where $H' = (H, \gamma\colon \tau \to \sigma)$. Since c is constrained, c′ has the form $f(c_1,\ldots,c_s)$ where $f \in \mathrm{NT}$ corresponds to $(\phi, t_1, t_2, s) \in \Phi$. Then $\tau = t_1(\tau_1,\ldots, \tau_s)$, $\sigma = t_2(\sigma_1,\ldots, \sigma_s)$, and for $j = 1, \ldots, s$, $(H' \vdash c_j\colon \tau_j \to \sigma_j)$.

Proceed by induction on k. Again, since c is constrained, $\mathrm{ht}(t_1) > 0$. $[\![\tau]\!] = t_1([\![\tau_1]\!],\ldots,[\![\tau_s]\!])$, so $[\![\tau]\!]$ has no elements of $\mathrm{ht} = 0$, the result holds for $k = 0$.

So assume the result for k and prove, for $\rho \in \mathrm{CEnv}(H, c, k{+}1)$ that $[\![c]\!].\rho$ maps $[\![\tau]\!]^{(k+1)}$ into $[\![\sigma]\!]$. By Prop. 5.1 we have

$$[\![c]\!].\rho = [\![c']\!].\rho[\gamma \mapsto [\![c]\!].\rho]$$

Set $\rho' = \rho[\gamma \mapsto [\![c]\!].\rho]$, and set $g_j = [\![c_j]\!].\rho'$, for each j. Then since $c' = f(c_1,\ldots, c_s)$,

$$[\![c']\!].\rho' = \phi(g_1, \ldots, g_s)$$

Claim $\rho' \in \mathrm{CEnv}(H', c_j, k)$, for $k = 1,\ldots,s$.

Proof. Take $\gamma' \in \mathrm{CVar}$ and closed M-terms ω_1 and ω_2. Suppose $\gamma' \in \mathrm{FV}(c_j)$ and $H'(\gamma') = (\omega_1 \to \omega_2)$. Of course, $\gamma' \in \mathrm{FV}(c')$. We must show that $\rho'(\gamma')$ maps $[\![\omega_1]\!]^{(k)}$ into $[\![\omega_2]\!]$. There are two cases.

If $\gamma' \neq \gamma$, then (i) $\gamma' \in$ FV(c), (ii) $\mathsf{H}(\gamma') = \mathsf{H}'(\gamma') = (\omega_1 \to \omega_2)$, (iii) $\rho(\gamma') = \rho'(\gamma')$. Since $\rho \in$ CEnv(H, c, k+1), we know that $\rho(\gamma')$ maps $[\![\omega_1]\!]^{(k+1)}$ into $[\![\omega_2]\!]$. But $[\![\omega_1]\!]^{(k)} \subseteq [\![\omega_1]\!]^{(k+1)}$, so this case is done.

If $\gamma'{=}\gamma$, then $\rho'(\gamma') = \rho'(\gamma) = [\![\mathsf{c}]\!].\rho$, and $\mathsf{H}'(\gamma') = \mathsf{H}'(\gamma) = (\tau \to \sigma)$. Thus we must show that $[\![\mathsf{c}]\!].\rho$ maps $[\![\tau]\!]^{(k)}$ into $[\![\sigma]\!]$. But $\rho \in$ CEnv(H, c, k+1) $\subseteq$ CEnv(H, c, k). So we are done by induction, and this proves our claim.

Now by the claim and induction, g_j maps $[\![\tau_j]\!]^{(k)}$ into $[\![\sigma_j]\!]$. Since ht(t_1) > 0, we can apply Proposition 3.8 with r = 1 to conclude that $\mathsf{t}_1(\overline{g})$ maps $(\mathsf{t}_1(\overline{[\![\tau]\!]})^{(k+1)}$ into $\mathsf{t}_1(\overline{[\![\sigma]\!]})$. Composing $\mathsf{t}_1(\overline{g})$ with $\phi(\overline{[\![\sigma]\!]})$: $\mathsf{t}_1(\overline{[\![\sigma]\!]}) \to \mathsf{t}_2(\overline{[\![\sigma]\!]})$, we get that $\phi(\overline{g})$ maps $(\mathsf{t}_1(\overline{[\![\tau]\!]}))^{(k+1)}$ into $\mathsf{t}_2(\overline{[\![\sigma]\!]})$, as required. $\square$

Remark 1 *This theorem holds with the slightly weaker restriction on compositions $c = c_2 \circ c_1$ that either c be closed, c_1 be trivial, or that c_2 be finite. But allowing c_2 to be finite causes problems in the proof of Theorem 7.3 below.*

7 Injective and invertible coercions

In this section we are interested in proving $[\![\mathsf{c}]\!]$ injective or bijective. This will require as above that c be constrained and closed, but we need as well assumptions on the behaviour of the primitive coercions and natural transformations occurring in c.

Definition 7.1 *We say c $\in$ CPrim is* injective *(resp.* surjective/bijective*) if $[\![\mathsf{c}]\!]$: $[\![A]\!] \to [\![B]\!]$ is injective (resp. surjective/bijective), where Σ(c) = (A$\to$B).*

Definition 7.2 *Let f $\in$ NT correspond to (ϕ, t_1, t_2, s) $\in \Phi$. We say f is injective (resp. surjective/bijective) if ϕ is injective (resp. surjective/bijective, ie, if for any $\overline{V} \in \mathcal{M}^s$, $\phi(\overline{V})$: $t_1(\overline{V}) \to t_2(\overline{V})$ is injective (resp. surjective/bijective).*

Notation 7.1 *For c $\in \mathcal{C}$, the set of primitive coercions occurring in c is written CPrim(c), and the set of f $\in$ NT occurring in c is written NT(c).*

We also need to focus on the coercion environments ρ that are well-behaved on the free variables of c. For the moment we consider only the injective case.

Definition 7.3 *Let H be a coercion type assignment, c a coercion such that (H $\vdash$ c: $\tau \to \sigma$).* CEnv_{inj}(H, c) *is the set of all $\rho \in$ CEnv(H, c) such that, for any $\gamma \in$ CVar and closed M-terms ω_1 and ω_2:*

$$\textit{if } \gamma \textit{ free in c and } H(\gamma) = (\omega_1 \to \omega_2) \textit{ then } \rho(\gamma) \textit{ is an injective map of } [\![\omega_1]\!] \textit{ into } [\![\omega_2]\!]$$

For k $\in \mathbb{N}$ we define CEnv_{inj}(H, c, k) *to be the set of all ρ in CEnv(H, c, k) satisfying the weaker condition that $\rho(\gamma)$ maps $[\![\omega_1]\!]^{(k)}$ injectively into $[\![\omega_2]\!]$.*

Theorem 7.1 *Let c: $\tau \to \sigma$ be a typed coercion. If CPrim(c) contains only injective coercions and if NT(c) contains only injective natural transformations, then*

1. *for any natural number k and $\rho \in$ CEnv$_{inj}$(H, c, k), $[\![c]\!].\rho$ is injective on $[\![\tau]\!]^{(k)}$.*

2. *for any $\rho \in$ CEnv$_{inj}$(H, c), $[\![c]\!].\rho$ is injective on $[\![\tau]\!]$.*

3. *if c is closed, $[\![c]\!]$ is injective on $[\![\tau]\!]$.*

Proof. The other two assertions follow from the first. Its proof is by induction on the structure of c. We show only the case of a recursion, so suppose c = rec γ. c$'$, where c$'$ = f($\overline{c}$) and f $\in$ NT corresponds to (ϕ, t_1, t_2, s) with ϕ injective. We have (H$'$ $\vdash$ c$'$: $\tau \to \sigma$) where H$'$ = (H, γ: $\tau \to \sigma$). For j = 1, ..., s, there are closed M-terms τ_j and σ_j such that (H$'$ $\vdash$ c_j: $\tau_j \to \sigma_j$), $\tau = t_1(\overline{\tau})$, and $\sigma = t_2(\overline{\sigma})$. We have

$$[\![c]\!].\rho = [\![c']\!].\rho' = \phi([\![c_1]\!].\rho', \ldots, [\![c_s]\!].\rho')$$

where $\rho' = \rho[\gamma \mapsto [\![c]\!].\rho]$. We must show that $[\![c]\!].\rho$ is injective on $[\![\tau]\!]^{(k)}$ whenever $\rho \in$ CEnv$_{inj}$(H, c, k). Since t_1 has height > 0, the case k = 0 is vacuously true, so we proceed by induction on k. Assuming $\rho \in$ CEnv$_{inj}$(H, c, k+1), we will show that $\rho' \in$ CEnv$_{inj}$(H$'$, c, k). By induction (on c) we then know that each $[\![c_j]\!].\rho'$ is injective on $[\![\tau]\!]^{(k)}$. Since ϕ is injective, Prop 3.3 then shows that $[\![c]\!].\rho$ is also.

It remains to prove that $\rho' \in$ CEnv$_{inj}$(H$'$, c_j, k). So take γ' free in c_j and suppose H$'$(γ') = ($\omega_1 \to \omega_2$). We must show that $\rho'(\gamma')$ is injective on $[\![\omega_1]\!]^{(k)}$. If $\gamma' \neq \gamma$ then $\rho'(\gamma') = \rho(\gamma)$, H$'$($\gamma'$) = H($\gamma$), and c_j is free in c. Since $\rho(\gamma)$ is injective on $[\![\omega_1]\!]^{(k+1)}$, it is also injective on $[\![\omega_1]\!]^{(k)}$.

So suppose finally that $\gamma' = \gamma$. Then $\rho'(\gamma') = [\![c]\!].\rho$ and H$'$(γ') = ($[\![\tau]\!] \to [\![\sigma]\!]$), hence $\omega_1 = \tau$. Since $\rho \in$ CEnv$_{inj}$(H, c, k+1) $\subseteq$ CEnv$_{inj}$(H, c, k), we have (by induction on k) that $[\![c]\!].\rho$ is injective on $[\![\tau]\!]^{(k)}$. ☐

One can prove analogous results about surjective and bijective coercions, assuming surjectivity or bijectivity on CPrim(c) and NT(c). The statements would involve analogously restricted environments $\rho \in$ CEnv$_{surj}$(H, c, k), ... We leave this to the reader, remarking that the bijective case follows from the following discussion of formally invertible coercions.

7.1 Formally invertible coercions

In this section we show that if a constrained and closed coercion c is made up of (formally) invertible primitives and natural transformations, then c has an inverse c^{-1} (also closed and constrained), and we can construct c^{-1} formally.

For simplicity we assume henceforth that CPrim has no extraneous elements. More precisely, if c_1, $c_2 \in$ CPrim satisfy $[\![c_1]\!] = [\![c_2]\!]$, then $c_1 = c_2$. (We have already required that distinct elements in NT be semantically distinct by assuming a bijection NT $\leftrightarrow \Phi$.)

Definition 7.4 *We say c and d in CPrim are* formal inverses *if there exist A and B in $\mathcal{T}_{const}$ such that $\Sigma(c) = (A \to B)$, $\Sigma(d) = (B \to A)$, and moreover that $[\![c]\!]$: $[\![A]\!] \to [\![B]\!]$ and $[\![d]\!]$: $[\![B]\!] \to [\![A]\!]$ are inverse bijections. We say c is* formally invertible *if it has a formal inverse.*

We require not just that $[\![c]\!]$: $[\![A]\!] \to [\![B]\!]$ be bijective but that its inverse be denoted by some d $\in$ CPrim. Note that the formal inverse is unique if it exists, by our assumption that CPrim have no extraneous elements. We define CPrim_{inv} to be the set of formally invertible elements of CPrim, and denote the formal inverse of c by c^{-1}.

Definition 7.5 *We say that f and g $\in$ NT are* formal inverses *if f corresponds to (ϕ, t_1, t_2, s) $\in \Phi$ and g corresponds to (ψ, t_2, t_1, s) $\in \Phi$ such that ϕ and ψ are inverse isomorphisms (of functors). We say that f is* formally invertible *if it has a formal inverse.*

Again we note that formal invertibility is a stronger condition than that ϕ be an isomorphism of functors, and that the formal inverse is unique if it exists. We write NT_{inv} for the set of formally inverse elements of NT, and we denote the formal inverse of f $\in \text{NT}_{inv}$ by f^{-1}.

Now we extend our definition to arbitrary conversions:

Definition 7.6 *c $\in$ CExpr is* formally invertible *if each $c_1 \in$ CPrim(c) is formally invertible and also each f $\in$ NT(c) is formally invertible.*

Let CExpr_{inv} be the set of formally invertible coercions in CExpr. For c $\in \text{CExpr}_{inv}$, we define its formal inverse c^{-1} by replacing each occurrence in c of $c_1 \in$ CPrim by $(c_1)^{-1}$ and each occurrence in c of f $\in$ NT by f^{-1}, remembering to reverse the order in compositions. We have the following relations:

$$\gamma^{-1} = \gamma$$

$(\text{fold}_\sigma)^{-1} = \text{unfold}_\sigma$ and $(\text{unfold}_\sigma)^{-1} = \text{fold}_\sigma$ for σ a recursion

$$(\mathsf{c}_2 \circ c_1)^{-1} = (\mathsf{c}_1{}^{-1})\circ(\mathsf{c}_2{}^{-1})$$

$$(\mathsf{f}(\mathsf{c}_1, \ldots, \mathsf{c}_s))^{-1} = \mathsf{f}^{-1}((\mathsf{c}_1)^{-1}, \ldots, (\mathsf{c}_s)^{-1})$$

$$(\mathsf{rec}\ \gamma.\mathsf{c}')^{-1} = \mathsf{rec}\ \gamma.((\mathsf{c}')^{-1})$$

Clearly $(c^{-1})^{-1} = c$, and it is easy to see that $FV(c) = FV(c^{-1})$.

In proving that a constrained coercion c with $(H \vdash c\colon \tau \to \sigma)$ maps $[\![\tau]\!]$ into $[\![\sigma]\!]$, we used $\rho \in CEnv(H, c)$ to track assumptions about variables occurring free in c. In the current setting we must record assumptions about a pair of coercions which are formal inverses. To do so we need a pair of type assignments that reverse the source and destination in types assigned to the same variables, and we require an analogous condition on a pair of environments.

Definition 7.7 *The inverse of a type assignment H has the same domain but reverses the source and destination of each entry. We denote the inverse of H by H^{-1}. So $dom(H^{-1}) = dom(H)$, for any $\gamma \in dom(H)$, $H(\gamma) = (\omega_1 \to \omega_2)$ if and only if $H^{-1}(\gamma) = (\omega_2 \to \omega_1)$*

The proof of the following is straightforward.

Proposition 7.2 *Given $c \in CExpr_{inv}$ and a type assignment H, then $(H \vdash c\colon \tau \to \sigma)$ if and only if $(H^{-1} \vdash c^{-1}\colon \sigma \to \tau)$.*

Now for environments:

Definition 7.8 *Let c_1 and c_2 be formally inverse, and H a type assigment such that $(H \vdash c_1\colon \tau \to \sigma)$. We say (ρ_1, H_1, c_1) and (ρ_2, H_2, c_2) are* inversely related *if*

$$\rho_1.\gamma \text{ and } \rho_2.\gamma \text{ are inverse functions between } [\![\omega_1]\!] \text{ and } [\![\omega_2]\!]$$

whenever $\gamma \in FV(c_1)$ $(= FV(c_2))$ and $H(\gamma) = (\omega_1 \to \omega_2)$ (and hence also $H^{-1}(\gamma) = (\omega_2 \to \omega_1)$).

We make two observations:

1. It follows that $\rho_j \in CEnv(H_j, c_j)$ for $j = 1, 2$.

2. If c is formally invertible and closed, and H is a type assignment such that $(H \vdash\colon c\colon \tau \to \sigma)$, then for any ρ_1, ρ_2 in CEnv, we have that (ρ_1, H, c) and (ρ_2, H^{-1}, c^{-1}) are inversely related, since $FV(c) = \emptyset$.

Theorem 7.3 *Let c be a constrained and formally invertible coercion expression, and let H be a type assignment such that ($H \vdash c\colon \tau \to \sigma$). For ρ_1, $\rho_2 \in$ CEnv, if (ρ_1, H, c) and (ρ_2, H^{-1}, c^{-1}) are inversely related then*

$$[\![c]\!].\rho_1 : [\![\tau]\!] \to [\![\sigma]\!]$$

$$[\![c^{-1}]\!].\rho_2 : [\![\sigma]\!] \to [\![\tau]\!]$$

are inverse isomorphisms.

Corollary 7.4 *Let c be a closed, constrained, and formally invertible coercion of type ($\tau \to \sigma$). Then*

$$[\![c]\!] : [\![\tau]\!] \to [\![\sigma]\!] \text{ and}$$

$$[\![c^{-1}]\!] : [\![\sigma]\!] \to [\![\tau]\!]$$

are inverse bijections.

Proof. In the statement of the Corollary, we have written, as usual, $[\![c]\!]$ instead of $[\![c]\!].\rho$, which is independent of ρ, and similarly for c^{-1}. To prove the theorm, let c_1 = c and c_2 = c^{-1}. By symmetry it suffices to show that $([\![c_2]\!].\rho_2) \circ ([\![c_1]\!].\rho_1) = 1_{[\![\tau]\!]}$. As in the proof of the termination theorm, we will actually prove the stronger assertion that for each $k \in \mathbb{N}$

$$([\![c_2]\!].\rho_2) \circ ([\![c_1]\!].\rho_1) = 1_{[\![\tau]\!]} \text{ on } [\![\tau]\!]^{(k)}$$

provided that $\rho_2(\gamma) \circ \rho_1(\gamma) = 1_{[\![\omega_1]\!]}$ on $[\![\omega_1]\!]^{(k)}$ for any $\gamma \in FV(c_1)$, where $H(\gamma) = (\omega_1 \to \omega_2)$.

We proceed by induction on the structure of c_1. The assertion is true, with no assumptions on ρ_1 and ρ_2, if c_1 is primitive, or is a fold or unfold. If $c_1 = \gamma$, then $c_2 = \gamma$. $[\![c_k]\!].\rho_k = \rho_k(\gamma)$ for k = 1, 2, so the conclusion is an instance of the hypothesis.

If $c_1 = e_1 \circ d_1$, where ($H \vdash d_1\colon \tau \to \omega$) and ($H \vdash e_1\colon \omega \to \sigma$), then $c_2 = d_2 \circ e_2$ where $e_2 = (e_1)^{-1}$ and $d_2 = (d_1)^{-1}$. We have ($H^{-1} \vdash e_2\colon \sigma \to \omega$), ($H^{-1} \vdash d_2\colon \omega \to \tau$), and

$$([\![c_2]\!].\rho_2) \circ ([\![c_1]\!].\rho_1) = ([\![d_2]\!].\rho_2) \circ ([\![e_2]\!].\rho_2) \circ ([\![e_1]\!].\rho_1) \circ ([\![d_1]\!].\rho_1)$$

Now c_1 is constrained. If it is actually closed, then so are d_1 and e_1, so by induction the theorem holds for each (with no mention of k):

$$([\![d_2]\!].\rho_2) \circ ([\![d_1]\!].\rho_1) = 1_{[\![\tau]\!]} \text{ and } ([\![e_2]\!].\rho_2) \circ ([\![e_1]\!].\rho_1) = 1_{[\![\omega]\!]}$$

and so actually

$$([\![c_2]\!].\rho_2) \circ ([\![c_1]\!].\rho_1) = 1_{[\![\tau]\!]}$$

If c_1 is not closed then either d_1 or e_1 is trivial. If d_1 is trivial, so is d_2 and $[\![d_1]\!].\rho_1 = [\![d_2]\!].\rho_2 = 1_{[\tau]}$. The result follows since $[\![\tau]\!] = [\![\omega]\!]$ and by induction

$$([\![e_2]\!].\rho_2) \circ ([\![e_1]\!].\rho_1) = 1_{[\![\omega]\!]} \text{ on } [\![\omega]\!]^{(k)}$$

The argument is the same if e_1 is trivial.

Next suppose $c_1 = f(\overline{d})$ with $f \in NT$ corresponding to $(\phi, t_1, t_2, s) \in \Phi$. We have $\tau = t_1(\overline{\tau})$, $\sigma = t_2(\overline{\sigma})$, and $(H \vdash d_j\colon \tau_j \to \sigma_j)$ for $j = 1, \ldots, s$. Then $c_2 = g(\overline{e})$, where $g = f^{-1}$ corresponds to $<\psi, t_2, t_1, s> \in \Phi$ and ϕ and ψ are inverse isomorphisms. Each $e_j = (d_j)^{-1}$, and we have $(H^{-1} \vdash e_j\colon \sigma_j \to \tau_j)$. By induction we have

$$([\![e_j]\!].\rho_2) \circ ([\![d_j]\!].\rho_1) = 1_{[\![\tau_j]\!]} \text{ on } [\![\tau_j]\!]^{(k)}$$

for each j. Since $[\![c_1]\!].\rho_1 = [\![f(\overline{d})]\!].\rho_1 = \phi([\![d_1]\!].\rho_1, \ldots, [\![d_s]\!].\rho_1)$, and similarly $[\![c_2]\!].\rho_2 = \psi([\![e_1]\!].\rho_2, \ldots, [\![e_s]\!].\rho_2)$, we must show

$$\psi([\![e_1]\!].\rho_2, \ldots, [\![e_s]\!].\rho_2) \circ \phi([\![d_1]\!].\rho_1, \ldots, [\![d_s]\!].\rho_1)$$

agrees with $1_{[\![\tau]\!]}$ on $[\![\tau]\!]^{(k)}$. But this follows from Propositions 3.7 and 3.8 since

$$[\![\tau]\!]^{(k)} = (t_1(\overline{[\![\tau]\!]}))^{(k)} \subseteq t_1([\![\tau_1]\!]^{(k)}, \ldots, [\![\tau_s]\!]^{(k)})$$

Lastly suppose $c_1 = \text{rec}\ \gamma.\ f(\overline{d})$. The analysis is similar to the preceding except that we use the type assignments $H' = (H, \gamma\colon \tau \to \sigma)$ and $(H')^{-1} = (H^{-1}, \gamma\colon \sigma \to \tau)$, and furthermore the height condition on t_1 becomes essential. We have $f \in NT$ corresponding to $<\phi, t_1, t_2, s> \in \Phi$, $\tau = t_1(\overline{\tau})$, $\sigma = t_2(\overline{\sigma})$, and $(H' \vdash d_j\colon \tau_j \to \sigma_j)$ for $j = 1, \ldots, s$. Then $c_2 = \text{rec}\ \gamma.\ g(\overline{e})$, where $g = f^{-1}$ corresponds to $(\psi, t_2, t_1, s) \in \Phi$ and ϕ and ψ are inverse isomorphisms. Each $e_j = (d_j)^{-1}$, and we have $(H'_2 \vdash e_j\colon \sigma_j \to \tau_j)$. MESSED UP HERE

We evaluate, with $\rho'_1 = \rho_1[\gamma \mapsto [\![c_1]\!].\rho_1]$

$$[\![c_1]\!].\rho_1 = [\![f(d_1, \ldots, d_s)]\!].\rho'_1 = \phi([\![d_1]\!].\rho'_1, \ldots, [\![d_s]\!].\rho'_1)$$

and with $\rho'_2 = \rho_2[\gamma \mapsto [\![c_2]\!].\rho_2]$

$$[\![c_2]\!].\rho_2 = [\![g(e_1, \ldots, e_s)]\!].\rho'_2 = \psi([\![e_1]\!].\rho'_2, \ldots, [\![e_s]\!].\rho'_2)$$

Thus we want to show, for each natural number k, that

$$\psi([\![e_1]\!].\rho'_2, \ldots, [\![e_s]\!].\rho'_2) \circ \phi([\![d_1]\!].\rho'_1, \ldots, [\![d_s]\!].\rho'_1) \tag{1}$$

is the identity on $[\![\tau]\!]^{(k)}$, provided that

$$\rho_2(\gamma') \circ \rho_1(\gamma') = 1_{[\![\omega_1]\!]} \text{ on } [\![\omega_1]\!]^{(k)} \text{ for any } \gamma' \in FV(c_1) \cap FV(c_2) \tag{2}$$

where $\mathsf{H}_1(\gamma') = (\omega_1 \to \omega_2)$ (and hence also $\mathsf{H}_2(\gamma') = (\omega_2 \to \omega_1)$).

We prove (1) by induction on k. For k = 0, it is vacuously true: $[\![\tau]\!]^{(0)} = \emptyset$, since c_1 is constrained and hence height(t_1) > 0. So we show that the assertion (1) for k implies the same for k+1. We will show below that the hypothesis (2) at level k+1 lets us conclude inductively that $([\![e_j]\!].\rho_2') \circ ([\![d_j]\!].\rho_1')$ agrees with $1_{[\tau_j]}$ on $[\![\tau_j]\!]^{(k)}$, for each j. As in the previous case Proposition 3.7 then yields

$$\psi(\overline{[\![e]\!].\rho_2'}) \circ \phi(\overline{[\![d]\!].\rho_1'}) = 1_{[\![\tau]\!]} \text{ on } [\![\tau]\!]^{(k)}$$

This time we will apply Proposition 3.8 with r = 1. To do so, we observe that no path in t_1 of length less than 1 ends in a variable x_j. This follows from the constraint (on c_1) that height(t_1) > 0. We conclude that $[\![\tau]\!]^{(k+1)} = (t_1(\overline{[\![\tau]\!]}))^{(k+1)} \subseteq t_1(\overline{[\![\tau]\!]^{(k)}})$

We now justify the induction step. We assume (2) above at level k+1, and show the analogue at level k for the various e_j, d_j. Precisely, we show for any j and any $\gamma' \in \mathsf{FV}(e_j) \cap \mathsf{FV}(d_j)$, that $\rho_2'(\gamma') \circ \rho_1'(\gamma') = 1_{[\omega_1]}$ on $[\![\omega_1]\!]^{(k)}$, where $\mathsf{H}_1'(\gamma') = (\omega_1 \to \omega_2)$.

If $\gamma' \neq \gamma$, then $\mathsf{H}_1'(\gamma') = \mathsf{H}(\gamma)$ and $\rho_1'(\gamma') = \rho_1(\gamma)$, and $\rho_2'(\gamma') = \rho_2(\gamma$ so (2) at level k+1 gives the conclusion. And if $\gamma' = \gamma$, then $\mathsf{H}_1'(\gamma') = (\tau \to \sigma)$, $\rho_1'(\gamma') = [\![c_1]\!].\rho_1$, and $\rho_1'(\gamma') = [\![c_2]\!].\rho_2$. What we need then is

$$([\![c_2]\!].\rho_2) \circ ([\![c_1]\!].\rho_1) = 1_{[\![\tau]\!]}$$

on $[\![\tau]\!]^{(k)}$, which is just the induction hypothesis (1).

8 Our constraint is too restrictive

We now give an example of an unconstrained coercion which nevertheless defines a bijection between its source and destination M-types. This shows that our class of constrained coercions is too narrow.

8.1 Finding the coercion

Fix A and B in $\mathcal{T}_{const}$ and consider $\tau = \mu\ \alpha.(\mathsf{A}\times\mathsf{B} + \alpha)$ and $\sigma = \mu\ \beta.\mathsf{B} + \beta$. We seek a coercion from τ to $\mathsf{A}\times\sigma$. Repeated unfoldings show $[\![\tau]\!] = [\![\mathsf{A}\times\mathsf{B}]\!] + [\![\tau]\!] = [\![\mathsf{A}\times\mathsf{B}]\!] + ([\![\mathsf{A}\times\mathsf{B}]\!] + [\![\tau]\!]) = \ldots$. Unfolding σ gives $[\![\sigma]\!] = [\![\mathsf{B}]\!] + [\![\sigma]\!] = [\![\mathsf{B}]\!] + ([\![\mathsf{B}]\!] + [\![\sigma]\!]) = \ldots$, and so $[\![A\times\sigma]\!] = [\![A]\!]\times([\![B]\!] + ([\![B]\!] + ([\![B]\!] + \ldots)))$, in which we have "factored out" $[\![A]\!]$. The desired coercion is like an infinite distributive law except that the successive summands are ever more deeply nested.

We will use a coercion $f(1_A, 1_B, 1_\sigma)$ and abbreviate it as "factor", where $f \in NT$ corresponds to the "factor out" direction of the distributive law: $(\phi, y_1\times y_2 + y_1\times y_3, y_1\times(y_2 + y_3), 3) \in \Phi$. $\phi(\overline{V})$ maps $in_1(v_1, v_2)$ to $(v_1, in_1(v_2))$ and $in_2(v_1, v_3)$ to $(v_1, in_2(v_3))$. We have $(\vdash \text{factor}: A\times B + A\times\sigma \to A\times(B + \sigma)$.

Unfolding τ and σ, we would like c_1 satisfying $(\vdash c_1: A\times B + \tau \to A\times(B + \sigma))$. Then we can "multiply out" the destination M-term and take $c_1 = \text{factor} \circ c_2$ where c_2 goes from $A\times B + \tau$ to $A\times B + A\times\sigma$. Thus so far we have

$$c = (1_A + \text{fold}_\sigma) \circ \text{factor} \circ c_2 \circ \text{unfold}_\tau$$

We then take $c_2 = \text{rec}\ \gamma.1_{A\times B} + c_3$, where c_3 must satisfy $(H \vdash c_3: \tau \to A\times\sigma)$, with $H = (\gamma: A\times B + \tau \to A\times B + A\times\sigma)$. Retracing the same path, we try $c_3 = (1_A + \text{fold}_\sigma) \circ \text{factor} \circ c_4 \circ \text{unfold}_\tau$, where $(H \vdash c_4: A\times B + \tau \to A\times B + A\times\sigma)$. Now the recursion stops and we can take $c_4 = \gamma$. We summarize:

$$c = (1_A + \text{fold}_\sigma) \circ \text{factor} \circ c_2 \circ \text{unfold}_\tau$$

$$c_2 = \text{rec}\ \gamma.(1_{A\times B} + ((1_A + \text{fold}_\sigma) \circ \text{factor} \circ \gamma \circ \text{unfold}_\tau))$$

Note that c satisfies the constraint on recursions but not on compositions. We can forgive the terms $(1_A + \text{fold}_\sigma)$ and unfold_τ but $(\text{factor} \circ \gamma)$ violates the condition.

8.2 The induced functions

Fix an arbitrary $\rho \in \text{CEnv}$, and set $h = [\![c_2]\!].\rho$. Since $(1_A + \text{fold}_\sigma)$ and unfold_τ are identity maps, we have $[\![c]\!].\rho = g \circ h$, where $g = ([\![\text{factor}]\!]).\rho$ is the obvious isomorphism from $[\![A\times B + A\times\sigma]\!]$ to $[\![A\times(B + \sigma)]\!]$. We will show that h is a total function everywhere on $[\![A\times B + \tau]\!]$ and defines a bijection $[\![A\times B]\!] + [\![\tau]\!] \to [\![A\times B]\!] + [\![A]\!]\times[\![\sigma]\!]$.

We have $h = [\![1_{A\times B} + ((1_A + \text{fold}_\sigma) \circ \text{factor} \circ \gamma \circ \text{unfold}_\tau)]\!].\rho_1$ where $\rho_1 = \rho[\gamma \mapsto h]$. Thus $h = 1_{[\![A\times B]\!]} + g \circ h$.

8.3 h is total and bijective

We will deduce from the recursive equation above that h maps V bijectively onto W, where $V = [\![A\times B]\!] + [\![\tau]\!]$ and $W = [\![A\times B]\!] + [\![A\times\sigma]\!]$. The proof is by induction on the height of $v \in V$. One needs to remember the following identities:

$$[\![A\times B]\!] = [\![A]\!]\times[\![B]\!]$$

$$[\![\tau]\!] = [\![A\times B]\!] + [\![\tau]\!]$$

$$[\![\sigma]\!] = [\![B]\!] + [\![\sigma]\!]$$

$v \in [\![\tau]\!]$ has the form $in_1(x)$, with $x \in [\![A\times B]\!]$ or the form $in_2(y)$ with $y \in [\![\tau]\!]$.

We first show that h is defined for all v and that $h(v) \in W$. Since v cannot have height 0, the induction is launched. h is defined at $in_1(x)$ and maps it to itself. It is defined at $v = in_2(y)$ provided $g \circ h$ is defined at y. By induction h is defined at y and $h(y) \in W$. Since g maps W onto $[\![A]\!]\times([\![B]\!] + [\![\sigma]\!]) = [\![A\times\sigma]\!]$, we see that $g \circ h$ is indeed defined at y and that $(g \circ h)(y) \in [\![A\times\sigma]\!]$. So finally h is defined at $in_2(y)$ and $h(in_2(y)) = in_2((g \circ h)(y)) \in in_2([\![A\times\sigma]\!]) \subseteq W$. This completes the induction.

Let's introduce some notation for the parts of V and W. Set $V_1 = [\![A\times B]\!]$, $V_2 = [\![\tau]\!]$, $W_1 = [\![A\times B]\!]$ $(= V_1)$, and $W_2 = [\![A\times\sigma]\!]$. h maps $in_1(V_1)$ bijectively onto $in_1(W_1)$, and it maps $in_2(V_2)$ into $in_2(W_2)$.

To show h injective we consider v_1 and $v_2 \in V$ such that $h(v_1) = h(v_2)$, and show that $v_1 = v_2$. This is clear if the v_j are both in $in_1(V_1)$ and impossible if one is $in_1(V_1)$ and the other is not (since $in_1(W_1)$ and $in_2(W_2)$ are disjoint). So suppose $v_j = in_2(y_j)$ with $y_j \in V_j$, for j = 1, 2. Then $h(v_j) = in_2(\ g(h(y_j))\ )$. Since g is injective, we get $h(y_1) = h(y_2)$. Then by induction $y_1 = y_2$, and so $v_1 = v_2$.

For surjectivity we need the general form of elements in $[\![A\times B]\!] + [\![\tau]\!]$ and in $[\![A\times B]\!] + [\![A\times\sigma]\!]$. From $[\![\tau]\!] = [\![A\times B]\!] + [\![\tau]\!]$, it is easy to see that any element $t \in [\![\tau]\!]$ has the form

$$X(a, b, n) = in_2^n(in_1((a, b)))$$

for unique $a \in [\![A]\!]$, $b \in [\![B]\!]$, and $n \in \mathbb{N}$. (Here in_2^n is the n-th iterate of in_2.) Note that $X(a, b, n{+}1) = in_2(X(a, b, n))$.

Similarly from $\sigma = B + \sigma$ we get that any $s \in [\![\sigma]\!]$ has the form $in_2^n(\ in_1(b)\ )$ for unique $b \in [\![B]\!]$ and $n \in \mathbb{N}$, and so a general element of $W_2 = [\![A\times\sigma]\!]$ has the form

$$Y(a, b, n) = (a,\ in_2^n(\ in_1(b)\ ))$$

for unique $a \in [\![A]\!]$, $b \in [\![B]\!]$, and $n \in \mathbb{N}$. Thus a general element of $[\![A\times B]\!] + [\![A\times\sigma]\!]$ is either $in_1((a, b))$ or $in_2(Y(a, b, n))$.

Now we already have noted that h maps $in_1((a, b))$ to itself, so it will suffices to prove the following:

$$h(in_2(X(a, b, n))) = in_2(Y(a, b, n))$$

For n = 0,

$$
\begin{aligned}
\mathsf{h}(\mathsf{in}_2(\mathsf{X}(a, b, 0))) &= \mathsf{in}_2(\mathsf{g}(\mathsf{h}(\mathsf{X}(a, b, 0)))) \\
&= \mathsf{in}_2(\mathsf{g}(\mathsf{in}_1((a, b)))) \\
&= \mathsf{in}_2((a, \mathsf{in}_1(b))) \\
&= \mathsf{in}_2(\mathsf{Y}(a, b, 0))
\end{aligned}
$$

Assuming the result for n, we compute

$$
\begin{aligned}
\mathsf{h}(\mathsf{in}_2(\mathsf{X}(a, b, n+1))) &= \mathsf{in}_2(\mathsf{g}(\mathsf{h}(\mathsf{X}(a, b, n+1)))) \\
&= \mathsf{in}_2(\mathsf{g}(\mathsf{h}(\mathsf{in}_2(\mathsf{X}(a, b, n))))) \\
&= \mathsf{in}_2(\mathsf{g}(\mathsf{in}_2(\mathsf{Y}(a, b, n)))) && \text{// by induction} \\
&= \mathsf{in}_2(\mathsf{g}(\mathsf{in}_2(\ (a, \mathsf{in}_2^n(\mathsf{in}_1(b)))))) \\
&= \mathsf{in}_2((a, \mathsf{in}_2(\mathsf{in}_2^n(\mathsf{in}_1(b))))) \\
&= \mathsf{in}_2((a, \mathsf{in}_2^{n+1}(\mathsf{in}_1(b)))) \\
&= \mathsf{in}_2(\mathsf{Y}(a, b, n+1))
\end{aligned}
$$

This completes the proof that h is bijective.

9 Open questions

The notion of a constrained coercion as defined here suffices for the strategies used by the Mockingbird tool [5], and we have shown that some conditions on a well-typed coercion are needed. But the condition is not necessary, and it would be of interest to characterize more precisely the conditions for convergence.

A completeness question that remains is to characterize abstractly the set of partial functions denoted by well-typed coercion terms. The question of deciding coercibility is also open: whether for given τ and σ there exists a bijective (or injective) coercion between these terms. The report [3] posed a version of this question.

Finally we discuss the port M-type [2, 4, 5] which of great importance to Mockingbird but is not handled by the techniques of this report. Informally a port of type T is a reference to a channel to which one can submit values of type T. In the Mockingbird project ports are used not only to model message sending, but also RPC invocations and proxy references to functions and objects. Given a function with parameter types $T_1,\ldots,T_n$ and return type T_{rtn}, the type of an RPC invocation message is Invoke = $T_{parms} \times \text{port}(T_{rtn})$, where $T_{parms} = \times^n(T_1,\ldots,T_n)$. The port is intended to be used by the callee to send the return value. A proxy reference to the function above is modeled by port(Invoke), i.e., as a channel to which one can send RPC invocations. A proxy reference to a object (ignoring fields not represented by getter/setter methods) is then modeled as a sum of ports, one for each method.

The problem is how to provide a mathematical model for port(τ) and to integrate this with the theory of M-types and coercions presented in this report. Ports should be contravariant: given c of type $(\tau \rightarrow \sigma)$ we want port(c) to have type $(port(\sigma) \rightarrow port(\tau))$. The informal argument for this expectation is that if you can send a σ-value to a port, and if you also know how to coerce a τ-value to a σ-value, you can by composition send a τ-value.

A Expressions and initial algebras

In this report the formal languages we use are all term algebras, so we have viewed them as sets of abstract syntax trees rather than as sets of strings. Aside from avoiding discussions of parentheses and parsing, this point of view is more convenient for defining functions by structural induction. We gather here basic definitions and facts (see for example [12] or [10]).

We define a generic term algebra with a single binding operator and establish notations used for substitution. We then recall the basic fact that the set of terms is an initial algebra.

A.1 Expressions and Terms

Assume given a ranked alphabet $\mathcal{F}$ and a disjoint countably infinite set Var ("variables"). Let $\mathcal{F}_n$ denote the subset of elements of rank n (n a natural number). The set Expr($\mathcal{F}$) of expressions is the set of finite ordered trees whose nodes are labeled by elements of $\mathcal{F}$.

We suppose that $\mathcal{F}_1$ contains a set of symbols of the form b(x), one for each x $\in$ Var, and we let Binders be the set of all such symbols. We speak as usual of free and bound occurrences of a variable x in an expression e, and let FV(e) = {x $\in$ Var: x has a free occurrence in e}. We say that a variable z *appears* in an expression e if either z or b(z) labels a node of e. Following [12], we define *context substitution* {e/x}e_1, where e and e_1 are expressions and x is a variable, as the substitution of e for each free occurrence of x in e_1. This form of substitution is defined without renaming and does *not* ward off capture of free variables. For example, {y/x}(b(y).x) = b(y).y.

Alpha-equivalence is the smallest congruence $\approx$ on Expr($\mathcal{F}$) such that b(x).e $\approx$ b(y).{y/x}e for any expression e and variables x and y such that y doesn't appear in b(x).e. We denote alpha-equivalence by $\sim_\alpha$ and the set of alpha-equivalence classes by Term($\mathcal{F}$). We write $[e]_\alpha \in$ Term($\mathcal{F}$) for the class of e $\in$ Expr($\mathcal{F}$).

Given $[e]_\alpha$ and $[e_1]_\alpha$ it is always possible to find representatives e and e_1 such that for x in FV(e), there is no occurrence of b(x) in e_1. Then no capture happens in the context substitution $\{e/x\}e_1$, and we define $[[e]_\alpha/x][e_1]_\alpha = [\{e/x\}e_1]$. One can show that this term depends only one the alpha-equivalence classes of e and e_1.

A.2 Expr($\mathcal{F}$) as initial algebra

Recall [10][3] that an $\mathcal{F}$-algebra is a set A on which the elements of $\mathcal{F}$ act. More precisely, there are given functions

$$f_A : A^n \to A$$

for each f $\in \mathcal{F}_n$, for any n. (So for f $\in \mathcal{F}_0$, f_A is just an element of A.) A *morphism* h: A $\to$ A′ of $\mathcal{F}$-algebras is a function satisfying

$$h(f_A(a_1, \ldots, a_n)) = f_{A'}(h(a_1), \ldots, h(a_n))$$

for any n and $(a_1, \ldots, a_n) \in A^n$. Expr($\mathcal{F}$) is the initial $\mathcal{F}$-algebra, and so a morphism from Expr($\mathcal{F}$) to any $\mathcal{F}$-algebra A is given uniquely by defining a structure of $\mathcal{F}$-algebra on A. We frequently omit subscripts and write f for f_A.

Definition A.1 *A* continuous *$\mathcal{F}$-algebra A is a cpo A together with continuous functions f_A: $A^n \to A$, for each f $\in \mathcal{F}$ and each n $\in \mathbb{N}$. A morphism of continuous $\mathcal{F}$-algebras is a morphism of $\mathcal{F}$-algebras which is also a continous function.*

B Fixed point semantics of recursive terms

We review here the use of least fixed-points to define the semantics of recursive expressions. See [12] for background on cpos and more details on the following.

B.1 Fixed-point semantics

We are given a cpo A, which we intend to be the space of "meanings" of closed expressions. We assume A is pointed (has a least element $\perp_A$), and that A has the structure of continuous $\mathcal{F}'$-algebra, where $\mathcal{F}' = \mathcal{F}$ - Binders - Var. In our examples, defining this structure on the candidate A is straightforward. We extend A to a cpo $\mathcal{D}$ which parametrizes

[3] we prefer the word "algebra" to Courcelle's "magma"

the effects of variables and then give $\mathcal{D}$ the structure of a continuous $\mathcal{F}$-algebra. We then obtain a semantic function $[\![\ ]\!]$ from Expr($\mathcal{F}$) to $\mathcal{D}$. It then turns out that for a closed expression e, $[\![e]\!]$ is actually in A, as desired.

An *environment* is a function ρ: Var $\to$ A. The set Env of environments can be viewed as the cpo of continuous functions [Var $\to$ A], where Var is viewed as a discrete cpo. Recall that the partial order on Env is given by defining $\rho_1 \sqsubseteq \rho_2$ to mean ρ_1(x) $\sqsubseteq$ ρ_2(x) (in A) for any x in Var. If D $\subseteq$ Env is directed, then for any x in Var, D(x) = $\{\rho$(x): $\rho \in$ D$\}$ is directed, and sup(D) $\in$ Env is given by sup(D)(x) = sup(D(x)). Env has a least element, the environment that maps each x to $\perp_A$.

The functions eval_x: Env $\to$ A, mapping ρ to ρ(x), are continuous. Given an environment ρ, a variable x, and an element a $\in$ A, we define ρ[x $\mapsto$ a] $\in$ Env by ρ[x $\mapsto$ a](x) = a, and ρ[x $\mapsto$ a](y) = ρ(y) if y $\in$ Var and y $\neq$ x. We call ρ[x $\mapsto$ a] an *update* of ρ.

An expression with free variables will have a meaning that depends on assigning values, i.e., elements of A, to the free variables. Thus the meaning is a function from Env to A. In order to carry out our fixed-point constructions, we actually require continuous functions. We set $\mathcal{D}$ = [Env $\to$ A]. We write d.ρ instead of d(ρ), for d $\in \mathcal{D}$ and $\rho \in$ Env. $\mathcal{D}$ is a pointed cpo, whose least element maps each environment to $\perp_A$. Our semantic map will be a morphism of $\mathcal{F}$-algebras Expr $\to \mathcal{D}$.

We first define an $\mathcal{F}'$-algebra structure on $\mathcal{D}$. For f $\in \mathcal{F}'_0 = \mathcal{F}_0$, let $f_\mathcal{D}$: Env $\to$ A map each ρ to f_A. For a non-binding f $\in \mathcal{F}_n$ with n > 0, define $f_\mathcal{D}$: $\mathcal{D}^n \to \mathcal{D}$ by

$$(f_\mathcal{D}(d_1, \ldots, d_n)).\rho = f_A(d_1.\rho, \ldots, d_n.\rho)$$

Continuity of $f_\mathcal{D}$ follows from the assumed continuity of f_A.

To give $\mathcal{D}$ the structure of $\mathcal{F}$-algebra, we first define $x_\mathcal{D}.\rho = \rho$(x) for x $\in$ Var. It remains to define continuous functions $(b(x))_\mathcal{D}$: $\mathcal{D} \to \mathcal{D}$, for each x in Var. For this we need the theory of least fixed-points. We recall that if B is a pointed cpo, and g: B $\to$ B is continuous, then there exists b in B such that g(b) = b. Such a b is called a fixed-point of g. In fact, g has a *least* fixed-point, denoted fix(g), ie, a fixed-point of g which is $\sqsubseteq$ b for any fixed-point b of g. fix(g) can be computed as the supremum of $\{g^n(\perp_B)$: n a natural number$\}$, where g^n is the n-th iterate of g. Taking least fixed-point is itself continuous [12]:

Proposition B.1 *Let B be a pointed cpo. fix: [B $\to$ B] $\to$ B is continuous.*

Now our $\mathcal{F}$-algebra structure on $\mathcal{D}$ is completed by defining, for d $\in \mathcal{D}$ and $\rho \in$ Env,

$$((b(x))_\mathcal{D}(d)).\rho = fix(a \mapsto d.\rho[x \mapsto a]) \text{ where a ranges over A}$$

To show that this definition suffices, we must check various continuity properties.

Proposition B.2 *For x $\in$ Var, the map (ρ, a) $\mapsto$ ρ[x $\mapsto$ a] is a continuous function from Env $\times$ A to Env.*

Proof. It suffices to check continuity in each variable separately. For ρ fixed, to show the induced map A $\to$ Env, ie from A to [Var $\to$ A], is continuous, it suffices to show its composition with eval_y is a continuous map A $\to$ A, for each y in Var. The composition is the map (a $\mapsto$ ρ[x $\mapsto$ a](y)). If x = y, this is the identity map (a $\mapsto$ a) and if x $\neq$ y it is the constant map (a $\mapsto$ ρ(x)), so in both cases the map is continuous.

For a in A fixed, the induced map is (ρ $\mapsto$ ρ[x $\mapsto$ a), from Env to itself, and again we consider the composed maps Env $\to$ A given by (ρ $\mapsto$ ρ[x $\mapsto$ a](y)), for y in Var. For x = y the map is (ρ $\mapsto$ a), a constant, and for x $\neq$ y, the map is (ρ $\mapsto$ ρ(y)) = eval_y. $\square$

Proposition B.3 *For x in Var, the map*

$$(d, \rho) \mapsto (a \mapsto d.\rho[x \mapsto a])$$

is a continuous map from $\mathcal{D}\times$Env to [A $\to$ A]. Hence the composed map

$$(d, \rho) \mapsto fix(a \mapsto d.\rho[\alpha \mapsto a])$$

is a continuous map from $\mathcal{D}$ to A, and its curried form

$$d \mapsto (\rho \mapsto fix(a \mapsto d.\rho[x \mapsto a]))$$

is a continuous map from $\mathcal{D}$ to itself.

Proof. The second assertion follows from the continuity of fix: [A $\to$ A] $\to$ A, and the third from the continuity of currying, so we only need to consider the first. From the proposition above, we see that the map h: ($\mathcal{D}\times$Env$\times$A) $\to$ ($\mathcal{D}\times$Env) given by ((d, ρ) $\mapsto$ (d, ρ[x $\mapsto$ a])), is continuous. The map apply: ($\mathcal{D}\times$Env) $\to$ A, sending (d, ρ) to d(ρ), is also continuous. So the composition apply $\circ$ h is continuous as well. $\square$

The third assertion of this last proposition assures us that

$$((b(x))_{\mathcal{D}}(d)).\rho = fix(a \mapsto d.\rho[x \mapsto a])$$

does define $(b(x))_{\mathcal{D}}$: $\mathcal{D} \to \mathcal{D}$, and our $\mathcal{F}$-algebra structure on $\mathcal{D}$ is completely defined. We thus have a unique morphism of $\mathcal{F}$-algebras $[\![\ ]\!]$: M-expr $\to$ $\mathcal{D}$. In summary,

Theorem B.4 . *Let A be a pointed cpo, $\mathcal{F}$ a ranked alphabet as above. and $\mathcal{F}' = \mathcal{F}$ - Binders - Var. Given on A the structure of* continuous *$\mathcal{F}'$-algebra there is a unique function $[\![\ ]\!]$: Expr($\mathcal{F}$)$\to$ $\mathcal{D}$ satisfying, for ρ in Env,*

1. *for f in $\mathcal{F}_0$, $[\![f]\!].\rho = f_A$*

2. *for x in Var, $[\![x]\!].\rho = \rho(x)$*

3. *for non-binding f in $\mathcal{F}_n$, $n > 0$, and expressions $e_1, \ldots, e_n$*

$$[\![f(e_1, \ldots, e_n)]\!].\rho = f_A([\![e_1]\!].\rho, \ldots, [\![e_n]\!].\rho)$$

4. *for x in Var and any expression e*

$$[\![b(x).e]\!].\rho = fix(a \mapsto [\![e]\!].\rho[x \mapsto a])$$

B.2 Properties of the semantic mapping

We list here various straightforward consequences of the definitions. First, the meaning of an expression depends only on its free variables.

Proposition B.5 *Let e be an expression and ρ_1, ρ_2 be environments. Suppose $\rho_1(x) = \rho_2(x)$ for any x in $FV(e)$. Then $[\![e]\!].\rho_1 = [\![e]\!].\rho_2$.*

It follows that if e is closed, $[\![e]\!]$: Env $\to$ A is constant, and so we may identify $[\![e]\!]$ with its constant value.

Proposition B.6 *Given x in Var and expressions e, e_1, suppose for any variable $y \in FV(e)$, the binder $b(y)$ does not occur in e_1. Then for any $\rho \in Env$*

$$[\![\{e/x\}e_1]\!].\rho = [\![e_1]\!].\rho[x \mapsto [\![e]\!].\rho].$$

From this we deduce that the meaning of an expression depends only on its alpha-equivalence class, so $[\![]\!]$ is really defined on Term$(\mathcal{F})$.

Proposition B.7 *For expressions e_1 and e_2, if $e_1 \sim_\alpha e_2$, then $[\![e_1]\!] = [\![e_2]\!]$.*

Consequently we can restate proposition B.6 using term substitution, with no extra hypotheses on free variables.

Proposition B.8 *For any $x \in$ Var, e, $e_1 \in$ Expr, and $\rho \in$ Env*

$$[\![[e/x]e_1]\!].\rho = [\![e_1]\!].\rho[x \mapsto [\![e]\!].\rho]$$

Next we list several proposition allowing us to simplify binders.

Proposition B.9 *If the variable x does not occur free in expression e, then*

$$[\![b(x).e]\!] = [\![e]\!]$$

Unfolding does not change the meaning of an expression:

Proposition B.10 *For $x \in Var$ and $e_1 \in Expr$*

$$[\![b(x).e_1]\!] = [\![[e/x]e_1]\!]$$

Unfolding makes no progress with "degenerate" terms b(x).x, so it's not surprising that they have "no" meaning.

Proposition B.11 *For any variable x, $[\![b(x).x]\!] = \bot_{\mathcal{D}}$.*

Proposition B.12 *Let C be a pointed cpo, h: $C \times C \to C$ a continuous map. Define g: $C \to C$ by g(c) = h(c, c) for $c \in C$, and f: $C \to C$ by f(c) = fix ($c' \to h(c, c')$). Then g and f are continuous and fix(f) = fix(g).*

Proof. g = $\triangle_C \circ$ h, where $\triangle_C$ is the diagonal map ($c \mapsto (c, c)$), so g is continuous. Let curry(h) = ($c \mapsto h_c$) be the curried form of h: $h_c(c') = h(c, c')$. curry(h): $C \to [C \to C]$ is continuous, and so is fix: $[C \to C] \to C$. We have f = fix $\circ$ curry(h), so f is continuous.

Let x be a fixed point of f. Then x= f(x) = fix(h_x) so x= $h_x(x)$. But $h_x(x)$ = h(x, x) = g(x), so x is also a fixed point of g. From {fixed points of f} $\subseteq$ {fixed points of g} it follows that fix(g) $\sqsubseteq$ fix(f).

Now let $x \in C$ be a fixed point of g. f(x) = fix(h_x) by definition. Since x = g(x) = h(x, x) = $h_x(x)$, x is a fixed point of h_x and so x $\sqsubseteq$ f(x). Then from $\bot_C \sqsubseteq$ x, we get $f^n(\bot_C) \sqsubseteq f^n(x) \sqsubseteq$ x, for any natural number n, and hence fix(f) $\sqsubseteq$ x Thus fix(f) is a lower bound of {fixed points of g}, and fix(f) $\sqsubseteq$ fix(g). □

Proposition B.13 *For distinct x, y, z $\in$ Var and e $\in$ Expressions (?) if z does not occur in e, then*

$$[\![b(x).\ b(y).\ e]\!] = [\![b(z).\ \{z/x\}\{z/y\}e]\!]$$

Proof. Fix $\rho \in$ Env. For c, $c' \in$ C, let h(c, c') = $[\![e]\!].\rho.[x \mapsto c, y \mapsto c']$. h: $C \times C \to C$ is continuous. We define f and g as in Proposition B.12 and compute

$$
\begin{aligned}
[\![b(x).b(y).e]\!].\rho &= \mathrm{fix}(c \mapsto [\![b(y).e]\!].\rho[x \mapsto c]) \\
&= \mathrm{fix}(c \mapsto \mathrm{fix}(c' \mapsto [\![e]\!].\rho[x \mapsto c,\ y \mapsto c'])) \\
&= \mathrm{fix}(c \mapsto \mathrm{fix}(c' \mapsto h(c, c'))) \\
&= \mathrm{fix}(f) \\
&= \mathrm{fix}(g) \\
&= \mathrm{fix}(c \mapsto h(c, c)) \\
&= \mathrm{fix}(c \mapsto [\![e]\!].\rho[x \mapsto c,\ y \mapsto c])
\end{aligned}
$$

Now $[\![b(z).\ \{z/x,\ z/y\}e]\!].\rho = \mathrm{fix}(c \mapsto [\![\ \{z/x,\ z/y\}e]\!].\rho[z \mapsto c])$, and

$$
\begin{aligned}
[\![\ \{z/x\}\{z/y\}e]\!].\rho[z \mapsto c] &= [\![\ \{z/y\}e]\!].\rho[z \mapsto c][x \mapsto [\![z]\!].\rho[z \mapsto c]] \\
&= [\![\ \{z/y\}e]\!].\rho[z \mapsto c][x \mapsto c] \\
&= [\![e]\!].(\rho[z \mapsto c,\ x \mapsto c])[y \mapsto [\![z]\!].\rho[z \mapsto c,\ x \mapsto c] \\
&= [\![e]\!].\rho[z \mapsto c,\ x \mapsto c,\ y \mapsto c]
\end{aligned}
$$

But $[\![e]\!].\rho[z \mapsto c,\ x \mapsto c,\ y \mapsto c] = [\![e]\!].\rho[z \mapsto c,\ x \mapsto c]$ since z does not occur in e. □

C Functors and natural transformations

We remind the reader of some basic definitions and refer to [12] for more background. For simplicity we restrict to small categories.

Definition C.1 *A* category $\mathcal{C}$ *is a pair (ob($\mathcal{C}$), arr($\mathcal{C}$)) of sets with the following additional data and conditions. The elements ob($\mathcal{C}$) are called the objects of $\mathcal{C}$ and the elements of arr($\mathcal{C}$) are called the arrows (or "morphisms") of $\mathcal{C}$.*

For each arrow f, we are given two objects called the source and target of f. We write $f{:}C_1 \to C_2$ to mean that C_1 is the source of f and C_2 is its target.

In addition, for each object C, we are given is a (unique) arrow 1_C: $C \to C$ called the identity *of C. And for each pair of arrows f: $C_1 \to C_2$ and g: $C_2 \to C_3$, there is given a composed arrow written g∘f: $C_1 \to C_3$. The following equalities are assumed to hold whenever the relevant compositions are defined:*

1. *h∘(g∘f) = (h∘g)∘f, i.e., composition is associative*
2. *$1_{C_2} \circ f = f$*
3. *$f \circ 1_{C_1}$*

The basic example used in this report is $\mathcal{M}$, whose objects are all the subsets of $\mathcal{S}$ and whose arrows are all functions between those objects. We also consider the categories whose objects are cpos and whose arrows are continuous functions between them. Categories can be composed in various ways. For example in Section 2 we considered $\mathcal{M}^n$ whose objects are all n-tuples $\overline{V}$ = (V_1,..., V_n) of objects in $\mathcal{M}$, and whose arrows are all n-tuples $\overline{a}$: $\overline{V}\to\overline{W}$ of arrows a_j: $V_j\to W_j$, with component-wise composition.

Definition C.2 *Given categories $\mathcal{C}$ and $\mathcal{D}$, a* functor *H: $\mathcal{C} \to \mathcal{D}$ is a pair of functions, (H_{ob}, H_{arr}), where H_{ob}: ob($\mathcal{C}$)$\to$ob($\mathcal{D}$) and H_{arr}: arr($\mathcal{C}$)$\to$arr($\mathcal{D}$). These functions must satisfy:*

1. *if u:$A \to B$ in $\mathcal{C}$ then $H_{arr}(u)$: $H_{ob}(A) \to H_{ob}(B)$ in $\mathcal{D}$.*
2. *$H_{arr}(v \circ u) = H_{arr}(v) \circ H_{arr}(u)$*
3. *$H_{arr}(id_c) = id_d$ if $d = H_{ob}(c)$*

It is customary to omit the subscripts and write simply H for either H_{ob} or H_{arr}.

Definition C.3 *Given functors t and t' from $\mathcal{C}$ to $\mathcal{D}$, a* natural transformation *ϕ: $t \to t'$ is a family of morphisms (in $\mathcal{D}$)*

$$\phi(c) : t(c) \to t'(c)$$

one for each object c of $\mathcal{C}$, such that for any morphism f: $c \to c'$ in $\mathcal{C}$, we have

$$\phi(c') \circ t(f) = t'(f) \circ \phi(c)$$

We will use the notation $\phi(f)$ for this composed arrow $t(c) \to t'(c')$.

Natural transformations can be composed in an obvious way. Given ϕ_1: $t_1 \to t_2$ and ϕ_2: $t_2 \to t_3$, we can define $\phi_2 \circ \phi_1$: $t_1 \to t_3$ by

$$(\phi_2 \circ \phi_1)(c) = (\phi_2(c)) \circ (\phi_1(c))$$

and for each functor t there is a natural transformation 1_t:t$\to$t which is an identity for composition.

For fixed categories $\mathcal{C}$ and $\mathcal{D}$, we can thus define a new "higher-level" category $\mathcal{D}^{\mathcal{C}}$ whose objects are the functors t: $\mathcal{C} \to \mathcal{D}$ and whose arrows are the natural transformations between such functors. $\mathcal{D}^{\mathcal{C}}$ is called a functor category, and the natural transformations are thus also called morphisms of functors.

Proposition C.1 *Given morphisms $\phi_1 \colon t_1 \to t_2$ and $\phi_2 \colon t_2 \to t_3$ of functors in $\mathcal{D}^{\mathcal{C}}$, and also morphisms $a_1 \colon d_1 \to d_2$ and $a_2 \colon d_2 \to d_3$ in $\mathcal{C}$, we have*

$$\phi_2(a_2) \circ \phi_1(a_1) = (\phi_2 \circ \phi_1)(a_2 \circ a_1)$$

The proof is immediate from the definitions. We will need the following application of this result.

Proposition C.2 *Suppose given a morphism $\phi_1 \colon t_1 \to t_2$ of functors in $\mathcal{D}^{\mathcal{C}}$, and also morphisms $a_0 \colon d_0 \to d_1$, $a_1 \colon d_1 \to d_2$, and $a_2 \colon d_1 \to d_2$ in $\mathcal{C}$. Suppose*

$$a_1 \circ a_0 = a_2 \circ a_0$$

Then

$$\phi(a_1) \circ t_1(a_0) = \phi(a_2) \circ t_1(a_0)$$

Proof. Let $\mathsf{I} = 1_{t_1}$, the identity of t_1. Then $\mathsf{t}_1(\mathsf{a}_0) = \mathsf{I}(\mathsf{a}_1) \circ \mathsf{t}_1(\mathsf{a}_0) = \mathsf{I}(\mathsf{a}_0)$. Applying the previous proposition we get

$$\begin{aligned} \phi(\mathsf{a}_1) \circ \mathsf{t}_1(\mathsf{a}_0) &= \phi(\mathsf{a}_1) \circ \mathsf{I}(\mathsf{a}_0) \\ &= (\phi \circ \mathsf{I})(\mathsf{a}_1 \circ \mathsf{a}_0) \\ &= (\phi \circ \mathsf{I})(\mathsf{a}_2 \circ \mathsf{a}_0) \\ &= \phi(\mathsf{a}_2) \circ \mathsf{I}(\mathsf{a}_0) \\ &= \phi(\mathsf{a}_2) \circ \mathsf{t}_1(\mathsf{a}_0) \end{aligned}$$

In the body of this paper, we need the following definitions. Let $\underline{Set}$ be the category whose objects are sets and whose arrows are ordinary functions.

Definition C.4 *Given functors t_1 and $t_2 \in \underline{Set}^{\mathcal{C}}$, we say that a morphism $\phi \colon t_1 \to t_2$ is injective (resp. surjective, bijective) iff for any object c of $\mathcal{C}$, the function $\phi(c) \colon t_1(c) \to t_2(c)$ is injective (resp. surjective, bijective).*

References

[1] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(4):575–631, 1993.

[2] J. Auerbach and M. C. Chu-Carroll. The Mockingbird system: A compiler-based approach to maximally interoperable distributed programming. Technical Report RC20718, IBM T. J. Watson Research Center, 1997.

[3] J. S. Auerbach, C. M. Barton, and M. Raghavachari. Type isomorphisms with recursive types. Technical Report RC21247, IBM T. J. Watson Research Center, 1998.

[4] Joshua Auerbach, Charles Barton, Mark Chu-Carroll, and Mukund Raghavachari. Mockingbird: Flexible stub compilation from pairs of declarations. In *1999 IEEE International Conference on Distributed Computing Systems*, July 1999.

[5] Joshua Auerbach, Charles Barton, and Mukund Raghavachari. Automated generation of flexible structure-base type coercions. Submitted to ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation, available from the authors.

[6] Joshua S. Auerbach and James R. Russell. The Concert Signature Representation: IDL as intermediate language. In *Proceedings of the 1994 ACM SIGPLAN Workshop on Interface Definition Languages*, January 1994.

[7] M. Brandt and F. Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *Proc. 3d Int'l Conf. on Typed Lambda Calculi and Applications (TLCA), Nancy, France, April 2-4, 1997*, volume 1210 of *Lecture Notes in Computer Science (LNCS)*, pages 63–81. Springer-Verlag, April 1997.

[8] R. Di Cosmo. Private communication.

[9] R. Di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Birkhauser, 1995.

[10] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[11] Joseph (Yossi) Gil. Subtyping arithmetical types, Apr. 1999. Unpublished Manuscript.

[12] Carl A. Gunter. *Semantics of Programming Languages*. The MIT Press, 1993.