ELSEVIER

Theoretical Computer Science 139 (1995) 69–113

Theoretical Computer Science

Fundamental Study

Strong categorical datatypes II: A term logic for categorical programming

J. Robin B. Cockett[a,*], Dwight Spencer[b]

[a] *Department of Computer Science, University of Calgary, Calgary, Alberta T2N 1N4, Canada*
[b] *Department of Computer Science and Engineering, Oregon Graduate Institute, Portland, OR 97291, USA*

Received September 1992; revised March 1994
Communicated by M. Nivat

Abstract

This paper lifts earlier category-theoretic results on datatypes to the level of an abstract language suitable for categorical programming language implementation. The earlier work built a strongly normalizing categorical combinator reduction system based entirely on functorial strength which allows the distribution of context to the interior of datatypes.

Here we declare inductive (initial) and coinductive (final) datatypes in a Hagino–Wraith style to provide an expressive computing environment. An inductive (resp. coinductive) datatype consists of a strong type-forming functor accompanied by (1) a collection of constructors (resp. destructors) and (2) a fold (resp. unfold) which is parametrized by state transformations to realize the appropriate universal property.

The high complexity of programming exclusively in categorical notation (combinators) warrants the development of a friendlier language isomorphic to the distributive categorical setting. In this paper such a term logic, which is also the programming language of the *charity* programming system, is described. This logic is first-order and is dictated directly by the underlying categorical semantics. It embodies primitive inductive and coinductive principles which reflect the uniqueness properties of the fold and unfold combinators. Several basic progam equivalences are demonstrated to illustrate how these inductive and coinductive principles can be used.

1. Introduction

In Part I [3] a first-order categorical setting was developed in which initial and final categorical datatypes with reasonable computational properties could be built. There the notion of strength was used to capture an important aspect of higher-order settings in a first-order manner: that all definable datatypes should allow the

* Corresponding author. Email addresses: robin@cpsc.ucalgary.ca and dwights@cse.ogi.edu.

SSDI 304-3975(94)00099-5

distribution of contextual data uniformly throughout their structure. This distribution is carried out by a parametrized family (natural transformation) of "explicit strength" morphisms that uniquely accompany each defined functor. These functors and their datatypes are deemed "strong" because of the well-known correspondence of explicit strengths to the enrichment structure of strong functors over a closed symmetric monoidal category [10].

In this paper we introduce a term logic for these strong datatypes to provide a logical view of the underlying categorical ideas. The resulting logic has a different perspective on induction as the inference rules are based on the uniqueness of the fold combinator rather than the principle of structural induction. In first-order logics, unlike higher-order logics, structural induction does not automatically guarantee the existence of fold terms, although their uniqueness can be derived when they are present. Here the primitive inductive principle of fold uniqueness axiomatically generates fold terms. The ability to generate fold terms is in harmony with the requirements of programming.

Furthermore, this primitive style of axiomatization can be applied equally well to coinductive datatypes via the uniqueness of the unfold combinator. Consequently, the term logic allows reasoning about datatypes such as streams and lazy trees in a first-order manner. This should be compared to the coinductive principles proposed in [15, 14] which involve bisimulation and higher-order constructions.

The term logic can be used as a template for building categorically based languages or directly as a programming language. The programming language *charity* is based upon this term logic. *Charity* underlines the widely recognized importance of the fold and unfold combinators (see [9, 13]) by enforcing their exclusive use and thereby introduces a new and disciplined style of programming. We motivate the development by including *charity* examples of elementary programs using common datatypes.

1.1. Datatypes

We recall from Part I that a categorical setting can be built for strong initial and final datatypes by selecting a category with finite products and successively adding to it combinators for each datatype as it is adjoined. The adjoining is accomplished by datatype declaration; the two declarative formats used by Charity [2] are shown below:

$$\begin{array}{ll} \text{data } L(A)\to C & \text{data } C\to R(A) \\ \quad = c_1 : E_1(A,C)\to C & \quad = d_1 : C\to E_1(A,C) \\ \quad\quad | \ \ldots & \quad\quad | \ \ldots \\ \quad\quad | c_n : E_n(A,C)\to C. & \quad\quad | d_n : C\to E_n(A,C). \end{array}$$

These forms of definition were introduced by Hagino [5]. The factorizer versions of the Hagino "left" ($L(A)$ appears as a domain) and "right" ($R(A)$ appears as a codomain) forms of datatype declaration have been reworked in a style similar to that suggested by Wraith [20]. Wraith's technique also allowed the translation of the resulting combinator rewritings into the polymorphic lambda calculus thereby passing along to the theory the calculus' strong normalization property.

The "left" declaration delivers a strong initial datatype-forming functor L in which $L(A)$ is a structure with parametric data of type A. L is strong because it arrives paired with a natural transformation

$$\theta^L_{A,X}: L(A) \times X \to L(A \times X)$$

– a strength – that distributes the environment or context X throughout the structure of $L(A)$. This allows the context to be accessible for processing by actions on parametric data in the structure. $L(A)$ is built from its component structures $E_i(A, L(A))$ by canonical constructors labeled "c_i" where $c_i: E_i(A, L(A)) \to L(A)$. Importantly, $L(A)$ is the categorical coproduct of the $E_i(A, L(A))$ via the c_i.

The "left" declaration asserts that any map from the new type $L(A)$ to a type C is to be determined uniquely by a set of n programmer-chosen maps from $E_i(A,C)$ to C. In this way, a new map that operates on $L(A)$ data structures, called a *fold factorizer*, becomes available by specifying what may be usefully thought of as n state transformations. These state transformations supply recursive behavorial actions for the fold factorizer to perform on components of an initial data structure.

The "right" declaration acts dually to also provide a type-forming functor possessing a strength. Here a set of canonical destructors $d_i: R(A) \to E_i(A, L(A))$ are spawned whereby $R(A)$ is the categorical product (or a record) of the $E_i(A, L(A))$. A destructor's role is to project the corresponding component (or field). The "right" declaration dictates that maps to the type $R(A)$ from a type C are to be determined uniquely by a collection of programmer-chosen state expansions from C to $E_i(A,C)$. Such an R data structure-producing program is called an *unfold factorizer*.

The labels c_i and d_i name only the canonical operations and should not be confused with the programmer-chosen maps. These constructors and destructors are always associated with their respective datatypes and are in one-to-one correspondence with the programmer-chosen maps used for building each factorizer.

The parametric data A will usually range over a power of the given datatype category. We have avoided writing a tuple of variables by viewing it as ranging over an arbitrary category.

In Fig. 1 we provide examples of common datatypes. Database trees, DBtree, include node data and leaf data of different structure. Lists and colists are, respectively, the least and greatest solutions to the same domain equations as are the datatypes NElist/coNElist (nonempty lists) and NEtree/coNEtree (nonempty trees). These codatatypes include the infinite structures as well as the finite structures. In contrast, infinite lists, *Inflist*, and infinite binary trees, *Inftree*, do not have any corresponding inductive datatype and are forced to have infinitary structure.

Booleans

data $Bool \longrightarrow C$
$= true : 1 \longrightarrow C$
$\mid false : 1 \longrightarrow C.$

Natural Numbers

data $Nat \longrightarrow C$
$= zero : 1 \longrightarrow C$
$\mid succ : C \longrightarrow C.$

Finite Lists

data $List(A) \longrightarrow C$
$= nil : 1 \longrightarrow C$
$\mid cons : A \times C \longrightarrow C.$

Finite Binary Trees

data $Btree(A) \longrightarrow C$
$= bleaf : A \longrightarrow C$
$\mid bnode : C \times C \longrightarrow C.$

Infinite Binary Trees

data $C \longrightarrow Inftree(A)$
$= node : C \longrightarrow A$
$\mid fork : C \longrightarrow (C \times C).$

Infinite Lists

data $C \longrightarrow Inflist(A)$
$= head : C \longrightarrow A$
$\mid tail : C \longrightarrow C.$

Finite Database Trees

data $\mathrm{DBtree}(A, B) \longrightarrow C$
$= dbleaf : A \longrightarrow C$
$\mid dbnode : B \times (C \times C) \longrightarrow C.$

Non-empty Trees

data $\mathrm{NEtree}(A) \longrightarrow C$
$= neleaf : A \longrightarrow C$
$\mid nebranch : A \times (C \times C) \longrightarrow C.$

Non-empty Lists

data $\mathrm{NElist}(A) \longrightarrow C$
$= neunit : A \longrightarrow C$
$\mid necons : A \times C \longrightarrow C.$

CoLists

data $C \longrightarrow coList(A)$
$= split : C \longrightarrow 1 + A \times C.$

Co-Non-empty Trees

data $C \longrightarrow \mathrm{coNEtree}(A)$
$= nenode : C \longrightarrow A$
$\mid nefork : C \longrightarrow 1 + (C \times C).$

Co-Non-empty Lists

data $C \longrightarrow \mathrm{coNElist}(A)$
$= nehead : C \longrightarrow A$
$\mid netail : C \longrightarrow 1 + C.$

Fig. 1. Datatype definitions.

1.2. Inductive programs

As an illustration of programming with inductive datatypes, consider the *charity* declaration of database trees above. This datatype's parametric data are held by the type variables A and B. The leaves have values of type A and the internal nodes have values of type B.

The component types of the database trees declaration can be unraveled as follows to see their structure as functors:

$$E_1((A, B), \mathrm{DBtree}(A, B)) = Fst(Fst(((A, B), \mathrm{DBtree}(A, B)))) = A$$

and

$$E_2((A,B),\mathrm{DBtree}(A,B))$$
$$= Prod(Snd(Fst(((A,B),\mathrm{DBtree}(A,B)))), Diag(Snd(((A,B),\mathrm{DBtree}(A,B))))$$
$$= B \times (\mathrm{DBtree}(A,B) \times \mathrm{DBtree}(A,B)),$$

where *Fst*, *Snd*, *Prod*, and *Diag* are the basic strong functors of projections, product and diagonalization.

We preview the term logic with a sample *charity* program. The progam accepts database trees with natural number leaves and boolean internal nodes, and computes the sum of all leaves of the tree t that are residing within a subtree not rooted by a boolean equal to false:

```
def leaf_sum(t)={|dbleaf: v⇒v
                 |dbnode: (b,(s1,s2))⇒{true⇒s1+s2
                                       |false⇒0
                                       }(b)
                 |}(t)
```

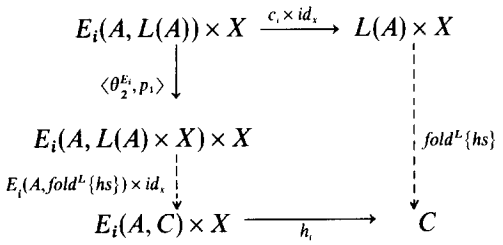
An example datum of the DBtree(*Nat*, *Bool*) data structure is

```
dbnode(true
      ,(dbnode(false
              ,(dbnode(true
                      ,(dbleaf(3)
                      ,(dbleaf(2)))
              ,dbnode(true
                      ,dbleaf(5)
                      ,dbleaf(0)))))
      ,dbnode(true
              ,(dbleaf(7)
              ,dbleaf(9)))).
```

In the program we presume previous declarations of the natural numbers, booleans, and definitions of basic functions such as '+'. The subprogram identified by the label "leaf:" processes only leaf substructures and does so by simply extracting the leaf's

A-value to form a partial result. The second subprogram, labeled by "node:", processes nodes by cases: if the node's boolean value is true the next partial result is computed by adding the partial results of the node's left and right subtrees contained in the result variables `s1` and `s2`; otherwise, the partial result is zero. The code body between the pair of decorated braces $\{|\ldots|\}$ is inductively driven by the structure of the tree `t` starting from the leaves. The conditional delimited by the $\{\ldots\}$ braces is a case analysis applied to the boolean argument `b`.

The earlier work [3] showed the fold factorizer morphism representing a program over an inductive datatype $L(A)$ to be the unique morphism $fold^L\{hs\}$ that must exist to make the collection of initiality diagrams below – one per constructor c_i – commute for any collection $hs = h_1, \ldots, h_n$ of morphisms:



The programmer's code specifies the action of the fold by supplying the state transformations h_i which may access context data of type X on the simpler component datatypes. Thus, the processing of a substructure $c_i(\_)$ by the right downward arrow is accomplished by going down the left arrows to distribute the context into the substructure and then process the substructure with contained context by applying the bottom transformation h_i. The inductive nature of the factorizer is exhibited by its appearance within the arrows on the left.

With these diagrams in mind, the reader may notice that context played no role in our first code example. The role of context can be illustrated by modifying that code to include an "access cost" overhead. For example, a leaf position may cost an extra `x1` and a node position an extra `x2` to access:

```
def leaf_sum(t, x1, x2) =

    {|leaf: v => v + x1

     |node: (b, t1, t2 => {true => (t1 + t2) + x2

                           |false => 0

                           }(b)

     |}(t)
```

The computation is again driven by the inductive structure of `t`: the auxillary data in the contextual variables `x1` and `x2` are now available everywhere within the program and in particular have been used to modify the transformations of the fold.

As the initiality diagrams completely define the datatype, $L(A)$ is an initial datatype. From them the general "recursive domain equation" for these initial datatypes.

$$L(A)\cong E_1(A,(L(A))+E_2(A,(L(A))+\cdots+E_n(A,L(A))$$

can be derived.

A special version of the initiality diagram defines the arrow component of the functor L, i.e. if $f:A\to B$ is a morphism between parameter datatypes, then $L(f):L(A)\to L(\mathrm{B})$ is definable as a fold factorizer by a careful selection of the h_i's (see Proposition 2.4). $L(f)$ in turn can be used to construct another important fold factorizer called the *map factorizer* of f, which is denoted $map^L(f)$. It inductively performs the action of f on each parametric datum – with context present – of the structure to produce a result of the same structure. Categorically, $map^L(f)$ can be thought of as the composition of (1) L's strength for diffusing the context into the structure and (2) the functoring of f to operate uniformly on the data-with-context, i.e. $map^L(f)\equiv\theta^L;\ L(f):L(A)\times X\to L(B)$.

Another fold factorizer, the *case factorizer* denoted as $case^L$, carries out a single operation via case analysis on an initial data structure to produce a result. The example node: subprogram illustrated this construct.

L's explicit strength $\theta^L_{A,X}:L(A)\times X\to L(A\times X)$ is also a fold factorizer and is therefore uniquely determined. The mysterious $\theta^{E_i}_2$ above is an abbreviation for an unstated morphism composition involving θ^{E_i} in which context is distributed only into the recursively defined parts of component structures.

The fold factorizer with its various specializations ($case^L$, map^L) can be used to provide a reasonable tool-kit for programming. They operate in the same manner as the like-named functions employed within conventional functional programming.

The rewriting rules for program evaluation can be extracted from the initiality diagrams of the specialized factorizers. The two possible paths in each diagram form the sides of the rule, with the rewriting direction towards the path involving the less complex component datatypes. The reductions (with some simplification) are:

$$c_i\times id_X;\ fold^L\{h_1,\ldots,h_n\}\Rightarrow\langle map^{E_i}\{p_0,fold^L\{h_1,\ldots,h_n\}\},p_1\rangle;\ h_i$$

$$c_i\times id_X;\ case^L\{h_1,\ldots,h_n\}\Rightarrow h_i$$

$$c_i\times id_X;\ map^L\{f_1,\ldots,f_m\}\Rightarrow map^{E_i}\{(f_1,\ldots,f_m),map^L\{f_1,\ldots,f_m\}\};c_i.$$

1.3. Co-inductive programs

While an inductive datatype typically serves to drive a program to process component-by-component the structure's basic data, a coinductive data structure is a result built starting from some seed datum. The coinductive components are independently constructed from the single seed by parallel subprograms.

The coinductive declaration delivers the finality diagrams below that make the functor R together with its destructors a final datatypes:

$$\begin{array}{ccc} C \times X & \xrightarrow{\langle g_i, p_1 \rangle} & E_i(A, C) \times X \\ & & \downarrow \theta_2^{E_i} \\ \textit{unfold}^R\{gs\} \;\vdots & & E_i(A, C \times X) \\ & & \downarrow E_i(A, \textit{unfold}^R\{gs\}) \\ R(A) & \xrightarrow[d_i]{} & E_i(A, R(A)) \end{array}$$

The diagrams express the programming requirements for producing a $R(A)$-typed result from a seed data value of type C in the presence of context X. The program $\textit{unfold}^R\{gs\}$ is determined by the subprogram $gs = g_1, \ldots, g_n$. These finality diagrams imply that $R(A)$ is isomorphic to the product of its component datatypes $E_i(A, R(A))$ (see [3]):

$$R(A) \cong E_1(A, (R(A)) \times E_2(A, (R(A)) \times \cdots \times E_n(A, R(A))$$

i.e. $R(A)$ can be regarded as a record and the destructors d_i act as field extractors.

As an example, consider the declaration below of binary search tress: Srtree(A, B). They are tree structures – possibly infinitary – having internal modes of datatype A and leaves of datatype B:

```
data C → Srtree(A, B) = srroot : C → A
                      | srbranch : C → B + (C × C).
```

The A-typed data occupies the internal nodes because the record pairs an A-value with either a single B-leaf or two search tree branches. The two destructors provided by the declaration have the respective typings

$\texttt{srroot} : R(A) \to A$

$\texttt{srbranch} : R(A) \to B + (R(A) \times R(A))$.

Below is a finite search tree example with boolean internal nodes and natural number leaves structured as a recursive record:

```
(srroot: true,
srbranch: bl((srroot: false
             ,srbranch: bl((srroot: false
                           ,srbranch: bO(7))
                           ,(srroot: true
                           ,srbranch: bO(5))))
             ,(srroot: true
             ,srbranch: bO(3))))
```

The b0 and b1 are the constructors of the sum datatype forming the codomain of srbranch.

With this example we again see the possibility of recursive definition, this time within a field. Here the srbranch: field contains either a basic datum of type B or recursively the same datatype within a product of two of its record structures. At the outermost level the record structure appears finitely as two fields labeled srroot: and srbranch: that tie a subtree root datum to its branches. Yet, internal to the srbranch: field an eventual infinitary structure may be implied by its specific subprogram if that subprogram builds the field in an open-ended recursive way. The disjoint sum forming the codomain of srbranch represents a choice at an internal node between selecting a terminating B-leaf value or continuing further a recursive construction of the srbranch: field. This implies that the infinitary subprogams must be invoked as-needed, or lazily, to produce only enough of a record sufficient for the computation at hand.

A sample program producing search trees with boolean nodes and natural number leaves is offered below. The assumed predicate divides is the usual number-theoretic one: $v_1 \mid v_2$. The (|...|) brackets delimit the program that is applied to the natural number n in the presence of the context x. The two subprograms that build the structure are srroot: and srbranch:. Notice that the resulting structure can be either finite or infinite depending on the values of n and x.

```
def div_tree(n, x) =

(|v ⇒ srroot: divides(v, x)

      |srbranch: {true ⇒ b0(v)

                  |false ⇒ b1(v − 1, v + 1)

                  }(divides(v, 2x))

 |)(n)
```

In the same manner as for initial datatypes, a final datatype $R(A)$'s unique explicit strength $\theta^R_{A,X}: R(A) \times X \to R(A \times X)$ is an unfold factorizer. Likewise, additional specializations of unfold factorizers are provided for programming convenience, viz. the record factorizer and the map factorizer.

The rewrite rules for computing with final datatypes are derived directly from the finality diagrams and appear below:

$$unfold^R\{g_1,\ldots,g_n\};d_i \Rightarrow \langle g_i, p_1\rangle; map^{E_i}\{p_0, unfold^R\{g_1,\ldots,g_n\}\}$$

$$record^R\{g_1,\ldots,g_n\};d_i \Rightarrow g_i$$

$$map^R\{f_1,\ldots,f_m\};d_i \Rightarrow d_i \times id_X; map^{E_i}\{(f_1,\ldots,f_m), map^R\{f_1,\ldots,f_m\}\}.$$

The standard set of product rewriting rules combined with the new rules accumulated by any finite set of datatype declarations form a confluent and terminating reduction system for evaluating programs that compute with the declared datatypes.

Also, the recursive nature of these rules motivates their corresponding initiality/finality diagrams being called *recursion diagrams.*

1.4. Outline

Section 2 introduces the term logic and discusses some of its basic properties. The design of the logic was motivated by its use as the programming language of *charity.* This logic has several novel features, notably the use of variable bases to provide a rudimentary level of pattern matching. As each inference rule is introduced we show its specialization to two specific datatypes. These datatype inference rules constitute the primitive induction and coinduction principles of this logic.

In Section 3 we illustrate how these rules can be used to establish some simple program equivalences. The proofs include the underlying categorical diagrams to illustrate the interplay between the term logic and the categorical combinators.

We describe in Section 4 how to establish the equivalence between the term logic and the underlying category theory. There are various ways of establishing this: we choose the brute force method by providing a pair of translations going in opposite directions and proving each is consistent and inverse to the other.

We have not included complete proofs. Many of the proofs proceed by a lengthy structural induction whose details are fascinating as exercises but numbing to the reader. The critical work lies in the setting up of the logics and translations; this we have presented in reasonable detail.

We present our concluding observations in Section 5.

2. Term logic

Coding solely with categorical combinators is an intimidating and unintuitive style of programming: raw combinators are overly detailed. This section develops a higher-level programming language as a term logic. The term logic eliminates the plenitude of projections occurring within combinator expressions: projections tend to proliferate as they are instrumental in the distribution of the context to the inside of datatypes. In Section 4 we show how the term logic corresponds exactly to the underlying distributive categorical structures.

We proceed by augmenting a "seed" finite product theory, corresponding to a category with finite products, with terms and rules that reflect the processing made available by the declaration of a sequence of strong datatypes. The term logic development generalizes the development for distributive categories in [1].

The term logic has been designed as a programming language. It is a foundation to which strong datatypes can be easily adjoined, and it can be translated or compiled into categorical combinators. It will emerge from the forthcoming definitions that the unit of translation is a term t in a context v. This is represented by a binding construct $\{v \mapsto t\}$ called an *abstraction.* This construct differs from λ-abstraction both superficially

by permitting pattern matching on products and semantically as it is not itself a term of the logic.

2.1. Finite product theories

A specification of a finite product (or many-sorted) theory, $\mathscr{D}=(T, F, S, E)$, consists of a set T of *primitive types*, a set F of *function symbols*, a *signature map* $S: F \to \mathscr{F}_{\times,1}(T) \times \mathscr{F}_{\times,1}(T)$, and a set E of *equations* between closed abstractions. The signature map S provides for each function symbol the domain and codomain types as free words on the primitive types. The full theory can then be generated by the following rules:

Types

The collection of *types*, $\mathscr{F}_{\times,1}(T)$, for the cartesian theory is defined inductively starting from the primitive types.

(i) if $\tau \in T$ then τ is a type.

(ii) 1 is a type.

(iii) If τ_0 and τ_1 are types then $\tau_0 \times \tau_1$ is a type.

Variables and variable bases

With every type τ there is assumed to be a countable collection of variables:

$$v_\tau, v_\tau^{(1)}, v_\tau^{(2)}, v_\tau^{(3)}, \ldots, v_\tau^{(i)}, \ldots.$$

For clarity, the superscripting will be omitted and the typing abbreviated, with both being inferable from discussion context. So v_i will typically mean $v_\tau^{(i)}$ for some infered type τ.

In addition to isolated variables, *variable bases* are provided as a programming convenience to carry out two tasks: (1) to bind more than one distinct variable at one time within a single expression and (2) to provide direct binding access (i.e. avoiding use of projections) to the components of data. These tasks are usually treated as part of the pattern matching process in functional languages.

Variable bases with their type associations are defined as follows.

(i) $()$ is variable base of type 1.

(ii) A variable v_τ is a variable base of type τ.

(iii) If v_0 and v_1 are variables bases of type τ_0 and τ_1, respectively, *having no variables in common* then (v_0, v_1) is a variable base of type $\tau_0 \times \tau_1$.

Terms

The variables, function symbols, and projections for product types are available for building terms. The inductive definition of terms and their associated types are given below.

(i) $()$ is a term of type 1.

(ii) A variable v_τ is a term of type τ.

(iii) If t_0 and t_1 are terms of type τ_0 and τ_1, respectively, then (t_0, t_1) is a term of type $\tau_0 \times \tau_1$.

(iv) If t is a term of the product type $\tau_0 \times \tau_1$ then $p_0(t)$ is a term of type τ_0 and $p_1(t)$ is a term of type τ_1.

(v) If f is any function symbol of F with $S(f)=(\tau, \tau')$, i.e. with domain of type τ and codomain of type τ', and t is a term of type τ, then $f(t)$ is a term of type τ'.

(vi) If t is a term of type τ, t' a term of type τ', and v a variable base of type τ' then $\{v \mapsto t\}(t')$ is a term of type τ.

It is worth remarking again that the abstraction $\{v \mapsto t\}$ above is *not* a term of the logic as functions are not to be terms in this first-order setting.

Free variables

Since we have a binding operator, viz. the abstraction, the terms have free and bound variables. The *free* variables of terms are determined inductively by the following.

(i) $fvas(())=\emptyset$.

(ii) If v is a variable, then $fvars(v)=\{v\}$.

(iii) $fvars((t_0, t_1))=fvars(t_0) \cup fvars(t_1)$.

(iv) If f is any function symbol (including projection), then $fvars(f(t))=fvars(t)$.

(v) $fvars(\{v \mapsto t\}(t'))=fvars(t') \cup (fvars(t)-fvars(v))$.

If an abstraction $\{v \mapsto t'\}$ occurs in a term t, the occurrence of a free variable in t' that occurs also in v becomes bound. A variable occurrence not judged bound is free. We shall conventionally refer to a variable as being *in* a term when that variable occurs freely in the term. Bound variables will be treated as invisible variables which can be renamed without changing the meaning of the term.

While in general an abstraction may have free variables, the *closed* abstractions, i.e. those having no free variables, are important as they are the "programs" of this setting. We say v is a *variable base* for t in the case $\{v \mapsto t\}$ is a closed abstraction.

Substitution

With the definitions of terms and free variables, we can inductively define the meaning of applying a simultaneous substitution to a term t'. We write the substitution as $\sigma_{v:=t}(t')$ where v is a variable base possessing the same type as the term t. The substitution involves matching the components of t to the variables of v in order to replace the occurrences of these variables in t'. The rules below describe precisely how this is done. Since variables may be substituted by terms containing variables, we hereafter assume for all substitution rules that renaming of bound variables, away from free variables is performed in parallel. The renaming prevents any variable clashes that would cause substituted free variables to be captured; doing it in parallel allows structurally inductive substitution proofs to be correctly built as demonstrated by Stoughton [18]. The rules are the following.

(i) $\sigma_{v:=t}(())=()$.

(ii) For each variable x.

(1) If x does not occur in v, then $\sigma_{v:=t}(x)=x$,
(2) if $x=v$ then $\sigma_{v:=t}(x)=t$,
(3) if $v=(v_0, v_1)$ and x occurs in v_i, then $\sigma_{v:=t}(x)=\sigma_{v_i:=p_i(t)}(x)$.

(iii) $\sigma_{v:=t}(t_0, t_1)=(\sigma_{v:=t}(t_0), \sigma_{v:=t}(t_1))$.
(iv) If f is any function symbol, then $\sigma_{v:=t}(f(t'))=f(\sigma_{v:=t}(t'))$.
(v) $\sigma_{v:=t}(\{v'\mapsto t'\}(t''))=\{v'\mapsto\sigma_{v:=t}(t')\}(\sigma_{v:=t}(t''))$.

Rules (ii) and (v) use the fact that bound variables are renamed away from free variables. Notice also that

$$\sigma_{(x,y):=(t_0,t_1)}(x, y)=(p_0(t_0, t_1), p_1(t_0, t_1)$$

and differs substitutively from

$$\sigma_{x:=t_0}(\sigma_{y:=t_1}(x, y))=(t_0, t_1)$$

due to the manner in which the substitutions is performed.

Axioms and inference rules

A set of axioms and rules are used to construct a type-indexed family of equivalence relations ($=_\tau$) among closed abstractions having terms of the same type and variable bases of type τ.

These relations are generated by the rules below. For brevity, $t_0 =_{v_\tau} t_1$ will be used to abbreviate $\{v\mapsto t_0\} =_\tau \{v\mapsto t_1\}$.

- *Equivalence of identities*: For all variable bases v and v' of type τ,

$$\{v\mapsto v\} =_\tau \{v'\mapsto v'\}.$$

- *Unit*

$$\frac{t \text{ is of type } 1 \qquad v \text{ is a variable base for } t}{t =_v ()}$$

- *Projection*

$$\frac{v \text{ is a variable base for } p_0(t_0, t_1)}{p_0(t_0, t_1) =_v t_0}$$

$$\frac{v \text{ is a variable base for } p_0(t_0, t_1)}{p_0(t_0, t_1) =_v t_1}$$

- *Application*:

$$\frac{v \text{ is a variable base for } \{v'\mapsto t\}(t')}{\{v'\mapsto t\}(t') =_v \sigma_{v':=t'}(t)}$$

- *Congruence*:

$$\frac{\{v_1\mapsto t_1\} =_\tau \{v_2\mapsto t_2\} \qquad \{v_1'\mapsto t_1'\} =_{\tau'} \{v_2'\mapsto t_2'\}}{\{v_1\mapsto\{v_1'\mapsto t_1'\}(t_1)\} =_\tau \{v_2\mapsto\{v_2'\mapsto t_2'\}(t_2)\}}$$

where t_1 and t_2 must be of type τ'.

The concept of *composing programs* can be expressed by the following definition where the types of v_1 and t_0 match:

$$\{v_0 \mapsto t_0\}; \{v_1 \mapsto t_1\} \equiv \{v_0 \mapsto \{v_0 \mapsto t_1\}(t_0)\}.$$

Its well-definedness up to $=_\tau$-equivalence is straightforwardly derivable from the congruence axiom.

The requirement of associativity up to equivalence then becomes

$$(\{v_0 \mapsto t_0\}; \{v_1 \mapsto t_1\}); \{v_2 \mapsto t_2\} = \{v_0 \mapsto t_0\}; (\{v_1 \mapsto t_1\}; \{v_2 \mapsto t_2\}).$$

where substitution is now forced by the composition rule to satisfy

$$\sigma_{v_1 := t_0}(\sigma_{v_2 := t_1}(t_2)) =_{v_0} \sigma_{v_2 := \sigma_{v_1 := t_0}(t_1)}(t_2).$$

This requirement is fulfilled as a corollary of the following elementary substitution property of the finite product theory.

Lemma 2.1 (Associativity of substitution). *Whenever* $\{v_2 \mapsto t_2\}$ *is a closed abstraction,*

$$\sigma_{v_1 := t_0}(\sigma_{v_2 := t_1}(t_2)) =_{v_0} \sigma_{v_2 := \sigma_{v_1 := t_0}(t_1)}(t_2).$$

With associativity we can also quickly derive the following lemma.

Lemma 2.2 (Simultaneity of substitution composition). *Whenever* $\{v_2 \mapsto t_2\}$ *is a closed abstraction,*

$$\sigma_{v_1 := t_0}(\sigma_{v_2 := t_1}(t_2)) =_{v_0} \sigma_{(v_2, v_1) := \sigma_{v_1 := t_0}(t_1), (t_0)}(t_2).$$

Note that the identity axiom could be replaced by a rule for explicitly renaming variables or the principle of extensionality,

$$\{v_1 \mapsto \{v_2 \mapsto t_2\}(v_1)\} =_\tau \{v_2 \mapsto t_2\}.$$

Furthermore, the surjective pairing identity

$$(p_0(t), p_1(t)) =_\tau t$$

is a consequence of the properties of substitution, making the type of t a finite product.

2.2. *Inductive datatype declarations*

Incrementing the theory by the declaration of a new inductive datatype

$$\begin{aligned}
&\text{data } L(A) \to C \\
&\quad = c_1 : E_1(A, C) \to C \\
&\qquad | \quad \ldots \\
&\qquad | c_n : E_n(A, C) \to C.
\end{aligned}$$

delivers the machinery to form and reason about the new terms and types introduced. This section describes the additional type, term, and inference rules introduced by such a declaration. The newly declared datatype must be built using types $E_i(A, L(A))$, $i=1,\ldots,n$, for the domains of the constructors which can be formed from the earlier-declared datatypes and the basic type forming rules.

In the description below we take the parametric arity of the functor of the datatype L to be m, i.e. $A=(A_1,\ldots,A_m)$.

Types

If $\tau_1,\ldots,\tau_m$ are types then $L(\tau_1,\ldots,\tau_m)$ is a type.

Variables and variable bases

$L(\tau_1,\ldots,\tau_m)$ has a countable collection $v^i_{L(\tau_1,\ldots,\tau_m)}$ of variables available each of which by itself forms a variable base.

Terms

(i) If t is a term of type $E_i((\tau_1,\ldots,\tau_m), L(\tau_1,\ldots,\tau_m))$ then $c_i(t)$ is a term of type $L(\tau_1,\ldots,\tau_m)$.

(ii) If t is a term of type $L(\tau_1,\ldots,\tau_m)$, v_i is a variable base of type $E_i((\tau_1,\ldots,\tau_m),\tau)$, and each t_i is a term of type τ for $i=1,\ldots,n$ then

$$\left\{\begin{bmatrix} c_1 : v_1 \mapsto t_1 \\ \cdots \\ c_n : v_n \mapsto t_n \end{bmatrix}\right\}(t)$$

is a term of type τ, viz. fold factorizer applied to t.

(iii) If t is a term of type $L(\tau_1,\ldots,\tau_m)$, v_i is a variable base of type $E_i((\tau_1,\ldots,\tau_m), L(\tau_1,\ldots,\tau_m))$, and each t_i is a term of type τ for $i=1,\ldots,n$ then

$$\left\{\begin{matrix} c_1(v_1)\mapsto t_1 \\ \cdots \\ c_n(v_n)\mapsto t_n \end{matrix}\right\}(t)$$

is a term of type τ, viz. a case factorizer applied to t.

(iv) If t is a term of type $L(\tau_1,\ldots,\tau_m)$, w_i is a variable base of the parameter type τ_i, and t_i is a term of type τ'_i for $i=1,\ldots,m$, then

$$L\begin{bmatrix} w_1\mapsto t_1 \\ \cdots \\ w_m\mapsto t_m \end{bmatrix}(t)$$

is a term of type $L(\tau'_1,\ldots,\tau'_m)$, viz. a map factorizer applied to t.

Each case/fold/map factorizer has a number of phrases each of which is an abstraction. These abstractions need not be closed and thereby permit the use of context variables in the abstracted term.

Free variables

The free variable rules are extended in the expected way for fold, case, and map factorizers:

(i) $fvars(c_i(t))=fvars(t)$.

(ii)

$$fvars(\left\{\begin{matrix} c_1 : v_1 \mapsto t_1 \\ \cdots \\ c_n : v_n \mapsto t_n \end{matrix}\right\}(t))=fvars(\left\{\begin{matrix} c_1(v_1)\mapsto t_1 \\ \cdots \\ c_n(v_n)\mapsto t_n \end{matrix}\right\}(t))=\left(\bigcup_{i=1}^{n}(fvars(\{v_i\mapsto t_i\}))\right)\cup fvars(t)$$

(iii)

$$fvars(L\begin{bmatrix} w_1\mapsto t_1 \\ \cdots \\ w_m\mapsto t_m \end{bmatrix}(t))=\left(\bigcup_{i=1}^{m}(fvars(\{w_i\mapsto t_i\}))\right)\cup fvars(t)$$

where $fvars(\{w_i\mapsto t_i\})=fvars(t_i)-fvars(w_i)$.

Substitution

(i) $\sigma_{v:=t}(c_i(t'))=c_i(\sigma_{v:=t}(t'))$.

(ii) $$\sigma_{v:=t}(\left\{\begin{matrix} c_1 : v_1 \mapsto t_1 \\ \cdots \\ c_n : v_n \mapsto t_n \end{matrix}\right\}(t'))=\left\{\begin{matrix} c_1 : v_1 \mapsto \sigma_{v:=t}(t_1) \\ \cdots \\ c_n : v_n \mapsto \sigma_{v:=t}(t_n) \end{matrix}\right\}(\sigma_{v:=t}(t'))$$

(iii) $$\sigma_{v:=t}(\left\{\begin{matrix} c_1(v_1)\mapsto t_1 \\ \cdots \\ c_n(v_n)\mapsto t_n \end{matrix}\right\}(t'))=\left\{\begin{matrix} c_1(v_1)\mapsto \sigma_{v:=t}(t_1) \\ \cdots \\ c_n(v_n)\mapsto \sigma_{v:=t}(t_n) \end{matrix}\right\}(\sigma_{v:=t}(t'))$$

(iv) $$\sigma_{v:=t}(L\begin{bmatrix} w_1\mapsto t_1 \\ \cdots \\ w_m\mapsto t_m \end{bmatrix}(t'))=L\begin{bmatrix} w_1\mapsto \sigma_{v:=t}(t_1) \\ \cdots \\ w_m\mapsto \sigma_{v:=t}(t_m) \end{bmatrix}(\sigma_{v:=t}(t'))$$

Axioms and inference rules:

The $=_\tau$ relations are enlarged by the rules below. Each rule is accompanied by the specific form for both finite lists, List(A), and database trees, DBtree(A, B), as declared in Fig. 1.

(i) *Fold-from-L*:

$$\left\{\begin{matrix} c_1 : v_1 \mapsto t_1 \\ \cdots \\ c_n : v_n \mapsto t_n \end{matrix}\right\}(c_i(t))=_v \{v_i\mapsto t_i\}(E_i\begin{bmatrix} \cdots \\ w_{A_i}\mapsto w_{A_i} \\ \cdots \\ w_L\mapsto \left\{\begin{matrix} c_1 : v_1 \mapsto t_1 \\ \cdots \\ c_n : v_n \mapsto t_n \end{matrix}\right\}(w_L) \end{bmatrix}(t))$$

List A:

$$\left\{\begin{matrix}\mathrm{nil}:()\mapsto t_1\\ \mathrm{cons}:(v_A,v_C)\mapsto t_2\end{matrix}\right\}(\mathrm{nil}()) =_v t_1$$

$$\left\{\begin{matrix}\mathrm{nil}:()\mapsto t_1\\ \mathrm{cons}:(v_A,v_C)\mapsto t_2\end{matrix}\right\}(\mathrm{cons}(a,l)) =_v \{(v_A,v_C),\mapsto t_2\}(a,\left\{\begin{matrix}\mathrm{nil}:()\mapsto t_1\\ \mathrm{cons}:(v_A,v_C)\mapsto t_2\end{matrix}\right\}(l))$$

DBtree(A, B):

$$\left\{\begin{matrix}\mathrm{dbleaf}:v_A\mapsto t_1\\ \mathrm{dbnode}:(v_B,(v_C,v'_C))\mapsto t_2\end{matrix}\right\}(\mathrm{dbleaf}(a)) =_v \{v_A\mapsto t_1\}(a)$$

$$\left\{\begin{matrix}\mathrm{dbleaf}:v_A\mapsto t_1\\ \mathrm{dbnode}:(v_B,(v_C,v'_C))\mapsto t_2\end{matrix}\right\}(\mathrm{dbnode}(b),(t,t'))) =_v$$

$$\{(v_B,(v_C,v'_C))\mapsto t_2\}(b,(\{\cdot\}(t),\{\cdot\}(t')))$$

(ii) *Case-from-L*:

$$\left\{\begin{matrix}c_1(v_1)\mapsto t_1\\ \cdots\\ c_n(v_n)\mapsto t_n\end{matrix}\right\}(c_i(t)) =_v \{v_i\mapsto t_i\}(t)$$

List A:

$$\left\{\begin{matrix}\mathrm{nil}()\mapsto t_1\\ \mathrm{cons}(v_A,v_{\mathrm{List}(A)})\mapsto t_2\end{matrix}\right\}(\mathrm{nil}()) =_v t_1$$

$$\left\{\begin{matrix}\mathrm{nil}:()\mapsto t_1\\ \mathrm{cons}:(v_A,v_{\mathrm{List}(A)})\mapsto t_2\end{matrix}\right\}(\mathrm{cons}(a,l) =_v \{(v_A,v_{\mathrm{List}(A)})\mapsto t_2\}(a,l)$$

DBtree(A, B):

$$\left\{\begin{matrix}\mathrm{dbleaf}(v_A)\mapsto t_1\\ \mathrm{dbnode}(v_B,(v_{\mathrm{DBtree}},v'_{\mathrm{DBtree}}))\mapsto t_2\end{matrix}\right\}(\mathrm{dbleaf}(a)) =_v \{v_A\mapsto t_1\}(a)$$

$$\left\{\begin{matrix}\mathrm{dbleaf}(v_A)\mapsto t_1\\ \mathrm{dbnode}(v_B,(v_{\mathrm{DBtree}},v'_{\mathrm{DBtree}}))\mapsto t_2\end{matrix}\right\}(\mathrm{dbnode}(b),(t,t'))) =_v$$

$$\{(v_B,(v_{\mathrm{DBtree}},v'_{\mathrm{DBtree}}))\mapsto t_2\}(b,(t,t'))$$

(iii) *Map-on-L*:

$$L\begin{bmatrix} w_1 \mapsto t_1 \\ \cdots \\ w_m \mapsto t_m \end{bmatrix}(c_i(t)) =_v c_i(E_i\begin{bmatrix} \cdots \\ w_i \mapsto t_i \\ \cdots \\ w_L \mapsto L\begin{bmatrix} \cdots \\ w_i \mapsto t_i \\ \cdots \end{bmatrix}(w_L) \end{bmatrix}(t))$$

List(A):

$$\mathrm{List}[v_A \mapsto t](\mathrm{nil}()) =_v \mathrm{nil}()$$

$$\mathrm{List}[v_A \mapsto t](\mathrm{cons}(a, l) =_v \mathrm{cons}(t(a), \mathrm{List}[v_A \mapsto t](l))$$

DBtree(A, B):

$$\mathrm{DBtree}\begin{bmatrix} v_A \mapsto t_1 \\ v_B \mapsto t_2 \end{bmatrix}(\mathrm{dbleaf}(a)) =_v \mathrm{dbleaf}(\{v_A \mapsto t_1\}(a))$$

$$\mathrm{DBtree}\begin{bmatrix} v_A \mapsto t_1 \\ v_B \mapsto t_2 \end{bmatrix}(\mathrm{dbnode}(b,(t,t'))) =_v$$

$$\mathrm{dbnode}(\{v_B \mapsto t_2\}(b), (\mathrm{DBtree}[\ldots](t), \mathrm{DBtree}[\ldots](t')))$$

(iv) *L Fold uniqueness*:

$$\frac{\forall_{i=1}^{n}\ \{v_L \mapsto t\}(c_i(z_i)) =_{(v,z_i)} \{v_i \mapsto t_i\}(E_i\begin{bmatrix} w_A \mapsto w_A \\ w_L \mapsto \{v_L \mapsto t\}(w_L) \end{bmatrix}(z_i))}{\{v_L \mapsto t\}(z) =_{(v,z)} \left\{\begin{bmatrix} c_1 : v_1 \mapsto t_1 \\ \cdots \\ c_n : v_n \mapsto t_n \end{bmatrix}\right\}(z)}$$

List(A):

$$\frac{h(\mathrm{nil}()) =_v t_1 \text{ and } h(\mathrm{cons}(a,l)) =_{(v,(a,l))} t_2(a, h(l))}{h(l) =_v \left\{\begin{bmatrix} \mathrm{nil} : () \mapsto t_1 \\ \mathrm{cons} : (v_A, v_C) \mapsto t_2(v_A, v_C) \end{bmatrix}\right\}(l)}$$

DBtree(*A*, *B*):

$$\frac{h(\mathrm{dbleaf}(a)) =_{(v,a)} t_1(a) \text{ and } h(\mathrm{dbnode}(b,(z_1,z_2))) =_{(v,(b,(z_1,z_2)))} t_2(b, h(z_1), h(z_2))}{h(t) =_{(v,a)} \left\{\begin{bmatrix} \mathrm{dbleaf} : v_A \mapsto t_1(v_A) \\ \mathrm{dbnode} : (v_A, (v_C, v_C')) \mapsto t_2(v_A, (v_C, v_C')) \end{bmatrix}\right\}(t)}$$

The axioms imply the rewrite rules listed in the introduction. The distribution of context is reflected by the fact that context variables can occur in all phrases of the factorizer. It is left to the reader to verify that the associativity of substitution is preserved when the theory is augmented by an inductive datatype.

We have the term logic analogue of the **X**-sum lemma [3].

Proposition 2.3. *The following inference rule holds in the augmented theory*:

$$\frac{h(c_i(v_i)) =_{(v,v_i)} t_i \text{ for } i=1,\ldots,n}{h(x) =_{(v,x)} \left\langle \begin{matrix} \cdots \\ c_i(v_i) \mapsto t_i \\ \cdots \end{matrix} \right\rangle (x) =_v t_1}$$

This proposition can be directly verified, but follows from the next observation.

The next result shows that the augmentation process could have been alternately expressed entirely in terms of the fold factorizer. The separate presentation of case/fold/map factorizers is desirable for programming not only because they are usefully expressive but also their specialized reductions are more efficient. This is particularly true for the case factorizer.

Proposition 2.4.

(i)

$$\left\langle \begin{matrix} \cdots \\ c_i(v_i) \mapsto t_i \\ \cdots \end{matrix} \right\rangle (v_L) =_{(v_L,v_X)} p_1\left(\left\langle \begin{matrix} \cdots \\ c_i : v_i \mapsto \{v_i \mapsto (c_i(v_i), t_i)\}\,(E_i \left[\begin{matrix} w_A \mapsto w_A \\ (w_L, w_C) \mapsto w_L \end{matrix} \right](v_i)) \\ \cdots \end{matrix} \right\rangle (v_L) \right)$$

(ii)

$$L\left[\begin{matrix} \cdots \\ w_j \mapsto t_j \\ \cdots \end{matrix} \right](v_L) =_{(v_L,v_X)} \left\langle \begin{matrix} \cdots \\ c_i : v_i \mapsto c_i(E_i \left[\begin{matrix} \cdots \\ w_j \mapsto t_j \\ \cdots \\ w_l \mapsto w_L \end{matrix} \right](v_i)) \\ \cdots \end{matrix} \right\rangle (v_L)$$

2.3. Co-inductive datatype declarations

This section describes the process of adding strong final datatypes; it parallels the development of the preceding section. Declaring a new coinductive datatype

$$\begin{aligned} &\text{data } C \to R(A) \\ &\quad = d_1 : C \to E_1(A, C) \\ &\quad\ \ | \ \cdots \\ &\quad\ \ | d_n : C \to E_n(A, C). \end{aligned}$$

provides the machinery to form and reason about the new terms and the types introduced. As before for inductive datatypes, the E_i's must be earlier declared. The resulting new types, terms, and inference rules are described below. The parametric arity of R is taken to be m so that $A=(A_1,\ldots,A_m)$.

Types

If $\tau_1,\ldots,\tau_m$ are types then $R(\tau_1,\ldots,\tau_m)$ is a type.

Variables and variable bases

$R(\tau_1,\ldots,\tau_m)$ has a countable collection $v^i_{R(\tau_1,\ldots,\tau_m)}$ of variables available, each of which forms a variable base.

Terms

(i) If t is a term of type $R(\tau_1,\ldots,\tau_m)$ then $d_i(t)$ is a term of type $E_i((\tau_1,\ldots,\tau_m),R(\tau_1,\ldots,\tau_m))$.

(ii) If t is a term of type C, v_C is a variable base of type C, and each t_i is a term of type $E_i((\tau_1,\ldots,\tau_m),C)$ then

$$\left\langle\!\!\left\langle \begin{matrix} & d_1:t_1 \\ v_C\mapsto & \cdots \\ & d_n:t_n \end{matrix} \right\rangle\!\!\right\rangle$$

is a term of type $R(\tau_1,\ldots,\tau_m)$, viz. an unfold factorizer applied to t.

(iii) If t_i is a term of type $E_i((\tau_1,\ldots,\tau_m),R(\tau_1,\ldots,\tau_m))$ for $i=1,\ldots,n$ then $(d_1:t_1,\ldots,d_n:t_n)$ is a term of type $R(\tau_1,\ldots,\tau_m)$, viz. a record.

(iv) If t is a term of type $R(\tau_1,\ldots,\tau_m)$, w_i is a variable base of type τ_i, and t_i is a term of type τ'_i for $i=1,\ldots,m$, then

$$R\begin{bmatrix} w_1\mapsto t_1 \\ \cdots \\ w_m\mapsto t_m \end{bmatrix}(t)$$

is a term of type $R(\tau'_1,\ldots,\tau'_m)$, viz. a map factorizer applied to t.

The phrases of an unfold term can be reconstituted as separate abstract maps with the common variable base.

Free variables

We show only the rule for the unfold term. The rules for the remaining destructor, record, and map terms are the obvious ones:

$$fvars\left(\left\langle\!\!\left\langle \begin{matrix} & d_1:t_1 \\ v_C\mapsto & \cdots \\ & d_n:t_n \end{matrix} \right\rangle\!\!\right\rangle(t)\right)=\left(\bigcup_{i=1}^{n} fvars(t_i)-fvars(v_C)\right)\cup fvars(t)$$

Substitution

Again we only show the rule for the unfold term. Recall that the variable capture problem is evaded by parallel renaming during substitution:

$$\sigma_{v:=t}\left(\left\langle v_C \mapsto \begin{matrix} d_1 : t_1 \\ \cdots \\ d_n : t_n \end{matrix} \right\rangle (t')\right) = \left\langle v_C \mapsto \begin{matrix} d_1 : \sigma_{v:=t}(t_1) \\ \cdots \\ d_n : \sigma_{v:=t}(t_n) \end{matrix} \right\rangle (\sigma_{v:=t}(t'))$$

Axioms and inference rules

The rules below are added to the $=_t$ relations. Each rule is accompanied by specific examples for the infinite lists and the nonempty lists as declared in Fig. 1.

(i) *Unfold-to-R*:

$$d_i\left(\left\langle v_C \mapsto \begin{matrix} d_1 : t_1 \\ \cdots \\ d_n : t_n \end{matrix} \right\rangle (t)\right) =_v E_i \left[\begin{matrix} w_A \mapsto w_A \\ w_C \mapsto \left\langle v_C \mapsto \begin{matrix} d_1 : t_1 \\ \cdots \\ d_n : t_n \end{matrix} \right\rangle (w_C) \end{matrix} \right] (\{v_C \mapsto t_i\}(t))$$

Inflist(A):

$$\text{head}\left(\left\langle v_C \mapsto \begin{matrix} \text{head} : t_1 \\ \text{tail} : t_2 \end{matrix} \right\rangle (c)\right) =_v \{v_C \mapsto t_1\}(c)$$

$$\text{tail}\left(\left\langle v_C \mapsto \begin{matrix} \text{head} : t_1 \\ \text{tail} : t_2 \end{matrix} \right\rangle (c)\right) =_v \left\langle v_C \mapsto \begin{matrix} \text{head} : t_1 \\ \text{tail} : t_2 \end{matrix} \right\rangle (\{v_C \mapsto t_2\}(c))$$

coNElist(A):

$$\text{nehead}\left(\left\langle v_C \mapsto \begin{matrix} \text{nehead} : t_1 \\ \text{netail} : t_2 \end{matrix} \right\rangle (c)\right) =_v \{v_C \mapsto t_1\}(c)$$

$$\text{netail}\left(\left\langle v_C \mapsto \begin{matrix} \text{nehead} : t_1 \\ \text{netail} : b_0() \end{matrix} \right\rangle (c)\right) =_v b_0()$$

$$\text{netail}\left(\left\langle v_C \mapsto \begin{matrix} \text{nehead} : t_1 \\ \text{netail} : b_1(t_2') \end{matrix} \right\rangle (c)\right) =_v b_1\left(\left\langle v_C \mapsto \begin{matrix} \text{nehead} : t_1 \\ \text{netail} : b_1(t_2') \end{matrix} \right\rangle (\{v_C \mapsto t_2'\}(c))\right)$$

(ii) *Record-to-R*:

$$d_i(d_1 : t_1, \ldots, d_n : t_n) =_v t_i$$

Inflist(A):

$$\text{head}(\text{head} : t_1, \text{tail} : t_2) =_v t_1$$

$$\text{tail}(\text{head} : t_1, \text{tail} : t_2) =_v t_2$$

CoNElist(A):

$$\text{nehead}(\text{nehead} : t_1, \text{netail} : t_2) =_v t_1$$

$$\text{netail}(\text{nehead} : t_1, \text{netail} : b_0()) =_v b_0()$$

$$\text{netail}(\text{nehead} : t_1, \text{netail} : b_1(t_2')) =_v b_1(t_2')$$

(iii) *Map-on-R*:

$$d_i(R\begin{bmatrix} w_1 \mapsto t_1 \\ \cdots \\ w_m \mapsto t_m \end{bmatrix}(t)) =_v E_i \begin{bmatrix} \cdots \\ w_i \mapsto t_i \\ \cdots \\ w_R \mapsto R\begin{bmatrix} w_1 \mapsto t_1 \\ \cdots \\ w_m \mapsto t_m \end{bmatrix}(w_R) \end{bmatrix}(d_i(t))$$

Inflist(A):

$$\text{head}(\text{Inflist}[v_A \mapsto t'](t)) =_v \{v_A \mapsto t'\}(\text{head}(t))$$

$$\text{tail}(\text{Inflist}[v_A \mapsto t'](t)) =_v \text{Inflist}[v_A \mapsto t'](\text{tail}(t))$$

coNElist(A):

$$\text{nehead}(\text{coNElist}[v_A \mapsto t'](t)) =_v \{v_A \mapsto t\}(\text{nehead}(t))$$

$$\text{netail}(\text{coNElist}[v_A \mapsto t'](t)) =_v \text{coNElist}[v_A \mapsto t'](\text{netail}(t))$$

(iv) *R unfold uniqueness*:

$$\frac{\forall_{i=1}^{n}\ d_i(\{v_C \mapsto t\}(z_i)) =_{(v,z_i)} E_i \begin{bmatrix} w_A \mapsto w_A \\ v_C \mapsto t \end{bmatrix}(\{v_C \mapsto t_i\}(z_i))}{\{v_C \mapsto t\}(z) =_{(v,z)} \left\langle\!\left| v_C \mapsto \begin{matrix} d_1 : t_1 \\ \cdots \\ d_n : t_n \end{matrix} \right|\!\right\rangle(z)}$$

Inflist(A):

$$\frac{\mathrm{head}(t(c)) =_v t_1(c) \quad \text{and} \quad \mathrm{tail}(t(c)) =_v t(t_2(c))}{t(c) =_v \left(\left\langle v_C \mapsto \begin{matrix}\mathrm{head}: t_1(v_C) \\ \mathrm{tail}: t_2(v_C)\end{matrix}\right\rangle\right)(c)}$$

coNElist(A):

$$\frac{\mathrm{nehead}(t(c)) =_v t_1(c) \text{ and } (\mathrm{netail}(t(c)) =_v b_0(( )) \text{ or } \mathrm{netail}(t(c)) =_v b_1(t(t'_2(c))))}{t(c) =_v \left(\left\langle v_C \mapsto \begin{matrix}\mathrm{nehead}: t_1(v_C) \\ \mathrm{netail}: t_2(v_C)\end{matrix}\right\rangle\right)(c)}$$

As expected, most of the counterparts of the strong initial datatype properties are operative for the final strong datatypes: associativity of substitution and definability of the strong final datatype record and map terms by means of the unfold term. These results are left to the reader to be found by imitating the techniques used to establish the corresponding initial datatype results.

3. Program equivalences

The term logic's validation with respect to the strong categorical datatype setting now lies in wait. Yet with the logic's inference rules afresh, this juncture is timely for demonstrating equivalences among programs. For the categorically inclined we have commuting diagrams to motivate proofs. It is, of course, the to-be-proved equivalence of term logic with the categorical setting which allows us to move freely back and forth between these two arenas.

The only induction principles available are the primitive forms provided by the fold-uniqueness and unfold-uniqueness rules. These should not be confused with the more familiar forms of structural induction and coinduction. As mentioned earlier, not only does structural induction demand the presence of more logic (e.g. quantification), but also in first-order settings it lacks the ability to generate fold and unfold factorizers.

The reader, with some justification, may question whether it is worth considering these apparently weaker forms of induction. The answer lies in the potential for automating a large class of proofs at this level. Interesting articles in this direction are [12, 17].

The game to be played in proving two programs, or closed abstractions, equal under the congruence of the term logic follows roughly the following pattern.

(i) Initially, attempt to reduce and/or transform both programs to a common form via factorizer reductions or previously proven program equivalences.

(ii) Otherwise, attempt to construct the recursion diagram for one of the programs to discover its defining specification in terms of the transformations t_i. Now allow the remaining program to play the role of t in the uniqueness rule and thereby use t and the t_i to construct the program equivalence subproblems that represent the rule's premises.

(iii) Repeat this method for all remaining generated subproblems.

Solving the subproblems produced by a pair of programs is equivalent to showing that one program behaves exactly the same as the other on common component structures of the datatype, i.e. both programs *share the same specification.*

In this section we shall not subscript the equality with a type or with variables provided the equality holds for all variable bases containing exactly the free variables of the terms.

3.1. *Associativity of appending lists*

With the definition

$$append(x, y) \equiv \left\{\begin{matrix} nil : () \mapsto y \\ cons : (a, c) \mapsto cons(a, c) \end{matrix}\right\}(x)$$

the equivalence to be shown is

$$append(x, append(y, z)) = append(append(x, y), z)).$$

Clearly, reductions are not applicable and previously proved transforms/equivalences are not yet available at this point, so we turn to constructing the recursion diagram for the program on the left, principally because it conveniently has a factorizer form

$$append(\_, append(y, z)) = \left\{\begin{matrix} nil : () \mapsto append(y, z) \\ cons : (a, c) \mapsto cons(a, c) \end{matrix}\right\}(\_)$$

We denote it below as pgm_l. The reader should note that strategic selection of the x-variable as the datatype variable, making the remaining variables serve as context variables. Abbreviating $List(A)$ as $L(A)$, we have

$$\begin{array}{ccccc} 1 \times (L(A) \times L(A)) & \xrightarrow{nil \times id} & L(A) \times (L(A) \times L(A)) & \xleftarrow{cons \times id} & (A \times L(A)) \times (L(A) \times L(A)) \\ \downarrow{\scriptstyle id} & & \downarrow{\scriptstyle pgm_l} & & \downarrow{\scriptstyle \langle ass^{-1}; id \times pgm_l, p_1 \rangle} \\ 1 \times (L(A) \times L(A)) & \xrightarrow[p_1; append(-,-)]{} & L(A) & \xleftarrow[p_0; cons]{} & (A \times L(A)) \times (L(A) \times L(A)) \end{array}$$

Using the program on the right as "$t(\_) = append(append(\_, y), z)$", we construct the premises of the fold uniqueness rule for $List(A)$. The two subprogram equivalence problems become

(i) $append(append(nil(), y), z) = append(y, z)$,

(ii) $append(append(cons(a, l), y), z) = cons(a, append(append(l, y), z))$.

The proof of (i) is immediate with a single factorizer reduction. The proof of (ii) requires two reductions:

$$\begin{aligned} append(append(cons(a,l),y),z) &= append(cons(a,append(l,y)),z) \\ &= cons(a,append(append(l,y),z)). \end{aligned}$$

Thus, the associativity property is established.

3.2. *Self-inversion of list reversal*

With the definition

$$reverse(x,y) \equiv \left\{ \begin{matrix} nil:()\mapsto nil() \\ cons:(a,c)\mapsto append(c,[a]) \end{matrix} \right\}(x),$$

where $[a]$ abbreviates $cons(a,nil())$, we wish to show $reverse(reverse(x))=x$. This time we draw the recursion diagram for the right-hand side, i.e. the identity:

$$\begin{array}{ccccc} 1 & \xrightarrow{nil} & L(A) & \xleftarrow{cons} & A\times L(A) \\ {\scriptstyle id}\downarrow & & \downarrow{\scriptstyle id} & & \downarrow{\scriptstyle id} \\ 1 & \xrightarrow[nil]{} & L(A) & \xleftarrow[cons]{} & A\times L(A) \end{array}$$

So to use fold-uniqueness, we let $t(\_)=reverse(reverse(\_))$, $t_1=nil()$ and $t_2(a,y)=cons(a,y)$, and thereby create the following subproblems:

(i) $reverse(reverse(nil()))=nil()$,

(ii) $reverse(reverse(cons(a,l)))=cons(a,reverse(reverse(l)))$.

The proof of (i) follows directly with two *reverse*-factorizer reductions. The example is completed with the proof of (ii) as

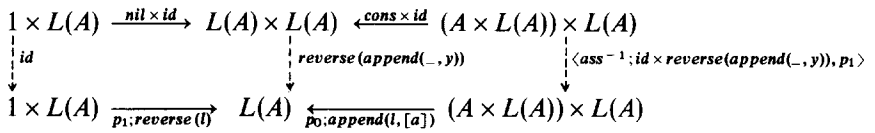
$$\begin{aligned} reverse(reverse(cons(a,l))) &= reverse(append(reverse(l),[a])) \\ &= append(reverse([a]),reverse(reverse(l))) \\ &= append([a],reverse(reverse(l))) \\ &= cons(a,reverse(reverse(l))) \end{aligned}$$

where the second step uses the following lemma.

Lemma 3.1.

$$reverse(append(x,y)) = append(reverse(y),reverse(x)).$$

Proof. The recursion diagram (with $L(A) \equiv List(A)$) for $reverse(append(\_, y))$ can be built as



The subproblems become

(i) $append(reverse(y), reverse(nil)) = reverse(y)$,

(ii) $append(reverse(y), reverse(cons(a, l))) = append(append(reverse(y), reverse(l)), [a])$.

Subproblem (i) proceeds as

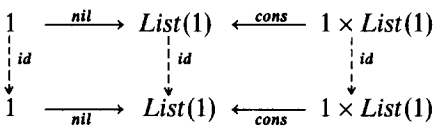
$$\begin{aligned} append(reverse(y), reverse(nil)) &= append(reverse(y), nil) \\ &= reverse(y), \end{aligned}$$

with the aid of a lemma ($append(x, nil) = x$) for t .e second step. The lemma quickly follows from a simple *id*-recursion diagram and is left to the reader. The derivation for (ii) comes via *append*'s associativity:

$$\begin{aligned} append(reverse(y), reverse(cons(a, l))) &= append(reverse(y), append(reverse(l), [a])) \\ &= append(append(reverse(y), reverse(l)), [a]). \quad \square \end{aligned}$$

3.3. *Reverse is identity in List*(1)

We try the obvious recursion diagram to attack the example, $reverse(x) = x$:

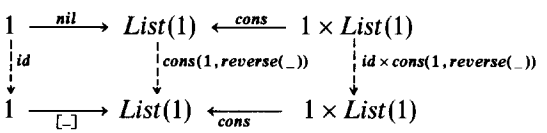


Therefore the following equivalences must be proved within *List*(1):

(i) $reverse(nil) = nil$,

(ii) $reverse(cons((), l)) = cons((), reverse(l))$.

The definition of *reverse* immediately yields (i). However, (ii) is considerably trickier – a plausible recursion diagram "candidate" for the right-hand side program is offered below:



The left square commutes by using one *reverse*-factorizer reduction. The right square commutes by the derivation below with the aid of a lemma for its third step:

$$\begin{aligned} cons(1, cons(( ), reverse(l))) &= cons(( ), append([( )], reverse(l))) \\ &= cons(( ), reverse(append(l, [( )]))) \\ &= cons(( ), reverse(append([( )], l))) \\ &= cons(( ), reverse(cons(( ), l))). \end{aligned}$$

The following lemma that generates the concerned step is the point at which our excursion depends precisely on all the elements of our lists being equal.

Lemma 3.2. *For any* $l \in List(1)$,

$$append(l, [( )]) = append([( )], l).$$

Proof. An appropriate recursion diagram for the left-hand program is

$$\begin{array}{ccccc} 1 & \xrightarrow{nil} & List(1) & \xleftarrow{cons} & 1 \times List(1) \\ \downarrow{\scriptstyle id} & & \downarrow{\scriptstyle append(\_,[( )])} & & \downarrow{\scriptstyle id \times append(\_,[( )])} \\ 1 & \xrightarrow[[\_]]{} & List(1) & \xleftarrow[cons]{} & 1 \times \mathrm{List}(1) \end{array}$$

Both squares are easily seen to be commuting by a single *append*-factorizer reduction. Thus, we need the equivalences

(i*) $append([( )], nil) = [( )]$

(ii*) $append([( )], cons(( ), l)) = cons(( ), append([( )], l))$

which are both also quickly attainable by *append*-factorizer reductions. □

The candidate diagram is appropriate for showing (ii). It brings forth the following subproblems for the left-hand side of (ii):

(i′) $reverse(cons(( ), nil)) = cons(( ), nil)$,

(ii′) $reverse(cons(( ), cons(( ), l))) = cons(( ), reverse(cons(( ), l)))$.

Completing the example, we see that (i′) is immediate and (ii′) is proved by our latest lemma:

$$\begin{aligned} reverse(cons(( ), cons(( ), l))) &= append(reverse(cons(( ), l)), [( )]) \\ &= append([( )], reverse(cons(( ), l))) \\ &= cons(( ), reverse(cons(( ), l))). \end{aligned}$$

It is straightforward to show a categorical isomorphism between the natural numbers datatype *Nat* and $List(1)$. This relationship warrants treating $append(x, y)$ for $x, y \in List(1)$ as the arithmetic addition of the natural numbers corresponding to x and y. With this interpretation it is interesting to note the consequent result, which does *not* hold for the polymorphic lambda calculus.

Corollary 3.3. *Addition is commutative*:

$$append(x, y) = append(y, x).$$

Proof. In *List*(1) we have, by the *reverse* acting as the identity,

$$\begin{aligned} append(x, y) &= reverse(append(x, y)) \\ &= append(reverse(y), reverse(x)) \\ &= append(y, x). \end{aligned}$$ □

3.4. Zipping streams makes a stream

This example illustrates primitive coinduction with an infinite list form of zipping. The problem is to show

$$zip(stream_1(c_1), stream_2(c_2)) = stream_{12}(c_1, c_2)$$

where the following definitions are applied:

$$stream_i(c_i) \equiv \left(\!\left| x \mapsto \begin{matrix} head : h_i(x) \\ tail : t_i(x) \end{matrix} \right|\!\right)(c_i) \quad (i = 1, 2)$$

$$stream_{12}(c_1, c_2) \equiv \left(\!\left| (x, y) \mapsto \begin{matrix} head : (h_1(x), h_2(y)) \\ tail : (t_1(x), t_2(y)) \end{matrix} \right|\!\right)(c_1, c_2)$$

$$zip(s_1, s_2) \equiv \left(\!\left| (x, y) \mapsto \begin{matrix} head : (head(x), head(y)) \\ tail : (tail(x), tail(y)) \end{matrix} \right|\!\right)(s_1, s_2)$$

The action of $stream_i$ on a seed value c_i creates an infinite list $\{h(c_i), h(t(c_i)), h(t^2(c_i)), h(t^3(c_i)), \ldots\}$. (For clarity, the field label and nested-parenthesis syntax requirements in the infinite list expressions are eased here in favor of list-like notation.) The program $stream_{12}$ operates on a pair of seed (values (c_1, c_2) to build an infinite list of derived pairs: $\{(h_1(c_1), h_2(c_2))), (h_1(t_1(c_1)), (h_2(t_2(c_2))), (h_1(t_1^2(c_1)), h_2(t_2^2(c_2))), \ldots\}$. The zip utility combines two infinite lists, or streams, of the form $\{c_1, c_2, c_3, \ldots\}$ and $\{c_1', c_2', c_3', \ldots\}$ into an infinite list of associated pairs: $\{(c_1, c_1'), (c_2, c_2'), (c_3, c_3'), \ldots\}$.

A reasonable recursion diagram for $stream_{12}$ is

$$\begin{array}{ccccc} A \times A & \xleftarrow{h_1 \times h_2} & C \times C & \xrightarrow{t_1 \times t_2} & C \times C \\ \Big\downarrow{\scriptstyle id} & & \Big\downarrow{\scriptstyle stream_{12}(\_,\_)} & & \Big\downarrow{\scriptstyle stream_{12}(\_,\_)} \\ A \times A & \xleftarrow[h_1 \times h_2]{} & Inflist(A \times A) & \xleftarrow[t_1 \times t_2]{} & Inflist(A \times A) \end{array}$$

The resulting program equivalences that must be verified are

(i) $head(zip(stream_1(c_1), stream_2(c_2))) = (h_1(c_1), h_2(c_2))$

(ii) $tail(zip(stream_1(c_1), stream(c_2))) = zip(stream_1(t_1(c_1)), stream_2(t_2(c_2)))$.

Unfold factorizer reductions easily provide both verifications. For (i) we have

$$\begin{aligned} head(zip(stream_1(c_1), stream_2(c_2))) &= (head(stream_1(c_1)), head(stream_2(c_2))) \\ &= (h_1(c_1), h_2(c_2)) \end{aligned}$$

and for (ii) we see

$$\begin{aligned} tail(zip(stream_1(c_1), stream_2(c_2))) &= zip(tail(stream_1(c_1)), tail(stream_2(c_2))) \\ &= zip(stream_1(t_1(c_1)), stream_2(t_2(c_2))). \end{aligned}$$

3.5. *Indexing infinite lists*

Two equivalent ways for indexed access of any infinite list built from a *fixed* pair $(h: C \times X \to A,\ t: C \times X \to C)$ of head and tail field-generating functions is presented. The example is interesting as it mixes primitive induction with unfolding.

First, we define a set of accessing routines relative to (h, t):

$$fetch(n, l) \equiv head\left(\left\{\begin{matrix} zero: () \mapsto l \\ succ: l' \mapsto tail(l') \end{matrix}\right\}(n)\right)$$

$$stream(c) \equiv \left(\left\langle x \mapsto \begin{matrix} head: h(x) \\ tail: t(x) \end{matrix}\right\rangle\right)(c)$$

$$expand(n, c) \equiv h\left(\left\{\begin{matrix} zero: () \mapsto c \\ succ: c' \mapsto t(c') \end{matrix}\right\}(n)\right)$$

The variables are typed with $n \in Nat$, $l, l' \in Inflist(A)$, and $c, c' \in C$. The program equality to be targeted here is

$$fetch(n, stream(c)) = expand(n, c).$$

The left-hand program accesses the nth entry of an infinite list built from the seed value c via the field builders h and t. The right-hand program first builds from c the tail-field value by applying t in succession the proper number of times and then applies the final head-field build operation. Essentially, *expand* accomplishes its work in ignorance of the $Inflist(A)$ datatype.

Some special observations are needed to find a path to the solution. First, the outermost functions of both programs (*fetch* on the left, *expand* on the right) are not quite in factorizer form due to the extra internal application of *head* and h, respectively, within them. So we define two simplications that *are* factorizers:

$$fetch^-(n, l) \equiv \left\{\begin{matrix} zero: () \mapsto l \\ succ: l' \mapsto tail(l') \end{matrix}\right\}(n)$$

$$expand^-(n, c) \equiv \left\{\begin{matrix} zero: () \mapsto c \\ succ: c' \mapsto t(c') \end{matrix}\right\}(n)$$

Now we rephrase our problem into a "factorizer" version. By observing that $head(stream(c)) = h(c)$ (an unfold factorizer reduction), we can strip off an outermost application of *head* from *both* sides of our proposed equivalence to obtain an alternative problem:

$$fetch^{-}(n, stream(c)) = stream(expand^{-}(n, c)).$$

The recursion diagram for the left-hand program with context variable c is a Nat-recursion as shown below:

$$\begin{array}{ccccc} 1 \times C & \xrightarrow{zero \times id} & Nat \times C & \xleftarrow{succ \times id} & Nat \times C \\ \Big\downarrow{\scriptstyle id} & & \Big\downarrow{\scriptstyle id \times stream; fetch^{-}} & & \Big\downarrow{\scriptstyle \langle id \times stream; fetch^{-}, p_1 \rangle} \\ 1 \times C & \xrightarrow{p_1; stream(\_)} & Inflist(A) & \xleftarrow{p_0; tail} & Inflist(A) \times C \end{array}$$

It remains only to show that $stream(expand^{-}(n, c))$ is defined by the same specification:

(i) $stream(expand^{-}(zero, c)) = stream(c)$,

(ii) $stream(expand^{-}(succ(n), c)) = tail(stream(expand^{-}(n, c)))$.

Subproblem (i) is a direct unfold factorizer reduction. The second one follows by applying to the left-hand program a fold factorizer reduction to the $expand^{-}$ subterm and a reverse unfold factorizer reduction to the result.

3.6. *Colists form a monoid*

To gather our intuition on colists recall that a colist can either by empty,

$$(split : b_0()),$$

or begin as

$$(split : b_1(a_1, (split : b_1(a_2, (split : b_1(a_3, (split : \ldots$$

and, if finite, end as

$$\ldots (split : b_0()) \ldots).$$

An append operation for colists can be expressed as follows:

$$coappend(x, y) \equiv \left\langle (l_1, l_2) \mapsto (split : \left\{ \begin{array}{l} b_0() \mapsto \left\{ \begin{array}{l} b_0() \mapsto b_0() \\ b_1(a, l) \mapsto b_1(a, (l_1, l)) \end{array} \right\} (split(l_2)) \\ b_1(a, l) \mapsto b_1(a, (l, l_2)) \end{array} \right\} (split(l_1))) \right\rangle (x, y)$$

The nested case analysis within *coappend* drives the colist generation firstly to replicate the first argument colist. If its replication completes, i.e. the first colist is finite, then replication continues by using the second colist. Note that the first colist argument, when infinite, dominates the *coappend* processing. If both arguments are

finite colists, the *coappend* produces results isomorphic to those of the familiar *List*-append operation.

The monoid properties to be proved are:

(i) $coappend(x,(split:b_0()))=x$,

(ii) $coappend((split:b_0()),x)=x$,

(iii) $coappend(x,coappend(y,z))=coappend(coappend(x,y),z)$.

The unfold uniqueness rule for $coList(A)$ to be leveraged for this purpose is

coList(A):

$$\frac{\left\{\begin{matrix} b_0()\mapsto b_0() \\ b_1(a,c)\mapsto b_1(a,t(c)) \end{matrix}\right\}(g(z))=_{(v,z)} split(t(z))}{t(c)=_v (\!|z\mapsto \mathrm{split}:g(z)|\!)(c)}$$

The right-unit property (i) is shown by using the finality diagram or the identity on augmented colists:

$$\begin{array}{ccc} coList(A) & \xrightarrow{split} & 1+A\times coList(A) \\ \Big\downarrow{\scriptstyle id} & & \Big\downarrow{\scriptstyle id+(id\times id)} \\ coList(A) & \xrightarrow[split]{} & 1+A\times coList(A) \end{array}$$

Consequently, employing unfold uniqueness for "t" as the left program of (i) and "$t_1(c)$" as $split(c)$ with respect to this diagram requries solving the subproblems:

(1) if $split(l)=b_1(t_1'(l),t_1''(l))$ then

$$split(coappend(l,(split:b_0())))=b_1(t_1'(l),coappend(t_1''(l),(slpit:b_0()))).$$

(2) if $split(l)=b_0()$ then

$$split(coappend(l,(split:b_0())))=b_0().$$

For showing (1), we make the indicated assumption for $split(l)$. Then we derive:

$$\begin{aligned} &split(coappend(l,(split:b_0()))) \\ &=E_1\left[\begin{matrix} w_A\mapsto w_A \\ (w_C,w_C')\mapsto coappend(w_C,w_C') \end{matrix}\right] \\ &\left(\left\{\begin{matrix} b_0()\mapsto \left\{\begin{matrix} b_0()\mapsto b_0() \\ b_1(a,l)\mapsto b_1(a,(l_1,l)) \end{matrix}\right\}(b_0()) \\ b_1(a,l)\mapsto b_1(a,(l,l_2)) \end{matrix}\right\}(split(l))\right) \\ &=E_1\left[\begin{matrix} w_A\mapsto w_A \\ (w_C,w_C')\mapsto coappend(w_C,w_C') \end{matrix}\right]\left(\left\{\begin{matrix} b_0()\mapsto b_0() \\ b_1(a,l)\mapsto b_1(a,(l,l_2)) \end{matrix}\right\}(b_1(t_1'(l),t_1''(l))\right) \\ &=E_1\left[\begin{matrix} w_A\mapsto w_A \\ (w_C,w_C')\mapsto coappend(w_C,w_C') \end{matrix}\right](b_1(t_1'(l),(t_1''(l),(split:b_0())))) \\ &=b_1(t_1'(l),coappend(t_1''(l),(split:b_0()))). \end{aligned}$$

The derivation for (2) is closely analogous to this one; it is left as an easy exercise. This completes the verification of the right-unit property. As expected, the proof of the left-unit property (ii) follows the same line of argument and is also left to the reader.

We finally arrive at the demonstration of associativity for *coappend*. The underlying *coList*-recursion diagram for the left-hand program of (iii) that motivates the proof is pictured as the outermost square of the diagram below:

$$\begin{array}{ccc}
coList(A)\times(coList(A)\times coList(A)) & \xrightarrow{t_1^\dagger} & 1+A\times(coList(A)\times(coList(A)\times coList(A))) \\
\downarrow{\scriptstyle id\times coappend} & & \downarrow{\scriptstyle id+id\times(id\times coappend)} \\
coList(A)\times coList(A) & \xrightarrow{t_1} & 1+A\times(coList(A)\times coList(A)) \\
\downarrow{\scriptstyle coappend} & & \downarrow{\scriptstyle id+id\times coappend} \\
coList(A) & \xrightarrow{split} & 1+A\times coList(A)
\end{array}$$

The bottom square commutes since it is the defining finality square for *coappend* where t_1 is the morphism corresponding to

$$\{(l_1,l_2)\mapsto\left\{\begin{array}{l} b_0()\mapsto\left\{\begin{array}{l} b_0()\mapsto b_0() \\ b_1(a,l)\mapsto b_1(a,(l_1,l))\end{array}\right\}(split(l_2)) \\ b_1(a,l)\mapsto b_1(a,(l,l_2))\end{array}\right\}(split(l_1))\}$$

The $t_1^\dagger$ morphism in the top square is the specification morphism for the left-hand program of (iii) and corresponds to

$$\{(l_1,(l_2,l_3))\mapsto$$

$$\left\{\begin{array}{l} b_0()\mapsto\left\{\begin{array}{l} b_0()\mapsto\left\{\begin{array}{l} b_0()\mapsto b_0() \\ b_1(a,l)\mapsto b_1(a,(l_1,(l_2,l)))\end{array}\right\}(split(l_3)) \\ b_1(a,l)\mapsto b_1(a,(l_1,(l,l_3)))\end{array}\right\}(split(l_2)) \\ b_1(a,l)\mapsto b_1(a,(l,(l_2,l_3)))\end{array}\right\}(split(l_1))\}$$

A straightforward diagram chase by case confirms the commutativity of the top square. The cases are

1. $l_1=b_1(a,l)$
2. $l_1=b_0()$ and $l_2=b_1(a,l)$
3. $l_1=l_2=b_0()$ and $l_3=b_1(a,l)$
4. $l_1=l_2=l_3=b_0()$.

For example, the chase for case 3 looks like

$$\begin{array}{ccc}
(l_1,(l_2,l_3)) & \xrightarrow{t_1^\dagger} & b_1(a,(l_1,l_2,l))) \\
\downarrow{\scriptstyle id\times coappend} & & \downarrow{\scriptstyle id+id\times(id\times coappend)} \\
(l_1,coappend(l_2,l_3))=(l_1,b_1(a,coappend(l_2,l))) & \xrightarrow{t_1} & b_1(a,(l_1,coappend(l_2,l)))
\end{array}$$

Thus, the diagram has been confirmed to be a recursion diagram for $coappend(x, coappend(y, z))$. Applying the unfold uniqueness rule for "$t(x,(y,z))$" equal to $coappend(coappend(x,y),z)$ and "t_1" equal to t_1^* yields the subproblems

(i') if $t_1^*(l_1,(l_2,l_3)) = b_1(t_1'(l_1,(l_2,l_3)), t_1''(l_1,(l_2,l_3)))$ then

$$split(coappend(coappend(l_1,l_2),l_3)) = b_1(t_1'(l_1,(l_2,l_3)), t(t_1''(l_1,(l_2,l_3))))$$

(ii') if $t_1^*(l_1,(l_2,l_3)) = b_0()$ then

$$split(coappend(coappend(l_1,l_2),l_3)) = b_0().$$

For proving (i'), we see that taking the indicated assumption for t_1^* forces consideration of the first three cases (1)–(3) for l_1, l_2 and l_3 used earlier in showing commutativity of the recursion diagram's top square. Because the derivations in these cases are alike, only the derivation for case 2 ($l_1 = b_1(), l_2 = b_0(a,l)$ is shown below. This case yields $t_1^*(l_1,(l_2,l_3)) = b_1(a,(l_1,(l,l_3)))$, which in turn says that $t_1'(l_1,(l_2,l_3)) = a$ and $t_1''(l_1(l_2,l_3)) = (l_1,(l,l_3))$. Proceeding, we have

$$\begin{aligned}
&split(coappend(coappend(l_1,l_2),l_3)\\
&= E_1[\ldots]\left(\left\{\begin{bmatrix} b_0() \mapsto \begin{Bmatrix} b_0() \mapsto b_0() \\ b_1(a,l) \mapsto b_1(a,(l_1,l)) \end{Bmatrix} \\ b_1(a,l) \mapsto b_1(a,(l,l_2)) \end{bmatrix}(split(l_3))\right\}(split(coappend(l_1,l_2)))\right)\\
&= E_1[\ldots]\left(\left\{\begin{bmatrix} b_0() \mapsto \begin{Bmatrix} b_0() \mapsto b_0() \\ b_1(a,l) \mapsto b_1(a,(l_1,l)) \end{Bmatrix} \\ b_1(a,l) \mapsto b_1(a,(l,l_2)) \end{bmatrix}(split(l_3))\right\}(b_1(a,(coappend(l_1,l),l_3)))\right)\\
&= E_1\begin{bmatrix} w_A \mapsto w_A \\ (w_C, w_C') \mapsto coappend(w_C, w_C') \end{bmatrix}(b_1(a,(coappend(l_1,l),l_3)))\\
&= b_1(a, coappend(coappend(l_1,l),l_3)).
\end{aligned}$$

Establishing (ii') requries case 4 in which all three arguments are empty, producing a simple case and map reduction. Thus, $coList(A)$ is a monoid.

3.7. Tree summations

Our last example is a comparison of two ways for summing the leaf values of a binary tree. The program *addtree* does it directly with a fold

$$addtree(t) \equiv \left\{\begin{bmatrix} bleaf : n \mapsto n \\ bnode : (n,m) \mapsto add(n,m) \end{bmatrix}\right\}(t)$$

A second way is to flatten the tree into a list of natural numbers and then sum the list. The appropriate routines are presented below:

$$flatten(t) \equiv \left\{\begin{matrix} bleaf : a \mapsto [a] \\ bnode : (l_1, l_2) \mapsto append(l_1, l_2) \end{matrix}\right\}(t)$$

$$sum(l) \equiv \left\{\begin{matrix} nil : () \mapsto zero() \\ cons : (n, m) \mapsto add(n, m) \end{matrix}\right\}(l)$$

The standard addition of natural numbers is defined by

$$add(n, m) \equiv \left\{\begin{matrix} zero : () \mapsto m \\ succ : v \mapsto succ(v) \end{matrix}\right\}(n)$$

Thus, the example problem under consideration is

$$sum(flatten(t)) = addtree(t).$$

From *addtree* being a fold factorizer, our problem translates to establishing the commutativity of the following diagram:

$$\begin{array}{ccccc} Nat & \xrightarrow{bleaf} & Btree(Nat) & \xleftarrow{bnode} & Btree(Nat) \times Btree(Nat) \\ \downarrow{\scriptstyle id} & & \downarrow{\scriptstyle sum(flatten(\_))} & & \downarrow{\scriptstyle sum(flatten(\_)) \times sum(flatten(\_))} \\ Nat & \xrightarrow{id} & Nat & \xleftarrow{add} & Nat \times Nat \end{array}$$

A *Btree*-fold reduction followed by a *List*-fold reduction validates the left square. For the right square, we note that

$$sum(flatten(bnode(t_1, t_2))) = sum(append(flatten(t_1), flatten(t_2)))$$

by a *Btree*-fold reduction. The desired result becomes immediate by showing

$$sum(append(l_1, l_2)) = add(sum(l_1), sum(l_2))$$

holds for any lists l_1 and l_2.

The left-hand side of the equation is the composition of two *List* fold factorizers: $append(\_, l_2)$ and $sum(\_)$. We apply the composition to the *List* cases to see if it can be expressed as a *List* fold factorizer:

$$sum(append(nil(), l_2)) = sum(l_2)$$

$$\begin{aligned} sum(append(cons(n, l), l_2)) &= sum(cons(n, append(l, l_2))) \\ &= add(n, sum(append(l, l_2))). \end{aligned}$$

Indeed, the composition is in fold factorizer form with *sum* occurring recursively exactly in accordance with its associated initiality diagram, viz. the composition is specified by the two maps $sum(l_2)$ and $add(n, sum(l'))$. We are left with applying the right-hand side of the lemma to the same cases to see whether it satisfies the same

specification. Using the easy-to-check associativity of *add*, we confirm below that the specification holds and that the right square commutes:

$$\begin{aligned} add(sum(nil()), sum(l_2)) &= add(zero(), sum(l_2)) \\ &= sum(l_2) \\ add(sum(cons(n, l)), sum(l_2)) &= add(add(n, sum(l)), sum(l_2)) \\ &= add(n, add(sum(l), sum(l_2))). \end{aligned}$$

The next section finally reveals that our successful examples are true rewards from this approach towards datatypes: the term logic is shown to have exactly the strong-functor foundation that was assumed and leveraged here for establishing program equivalences.

4. The equivalence of combinators and term logic

An equivalence between a categorical combinator theory and the term logic is expressed by a pair of mutually inverse consistent translations: combinators-to-programs and programs-to-combinators. Consistency simply means that a translation preserves equality.

A finite product combinator theory $\mathscr{C}$ has the specification $(\mathscr{T}, \mathscr{F}, \mathscr{S}, \mathscr{E})$ where $\mathscr{T}$ is a collection of primitive types, $\mathscr{F}$ is a collection of predetermined maps or combinators, $\mathscr{S}$ is the set of type signatures of the combinators, and $\mathscr{E}$ is a set of equations between combinators.

The types are given by the same rules used earlier for generating the cartesian theory types.

The combinators are generated inductively by the following.

- For every type τ there is an identity combinator $id_\tau : \tau \to \tau$.
- For every type τ there is a final combinator $!_\tau : \tau \to 1$.
- If $f \in \mathscr{F}$ has signature (τ_1, τ_2), then $f : \tau_1 \to \tau_2$ is a combinator.
- For every pair of types τ_0 and τ_1, there are projection combinators $p_0^{\tau_0,\tau_1} : \tau_0 \times \tau_1 \to \tau_0$ and $p_1^{\tau_0,\tau_1} : \tau_0 \times \tau_1 \to \tau_1$.
- If $c_0 : \tau \to \tau_0$ and $c_1 : \tau \to \tau_1$ are combinators, then their pairing $\langle c_0, c_1 \rangle : \tau \to \tau_0 \times \tau_1$ is a combinator.
- If $c_0 : \tau_0 \to \tau_1$ and $c_1 : \tau_1 \to \tau_2$ are combinators, then their composition $c_0 ; c_1 : \tau_0 \to \tau_2$ is a combinator.

The symmetric transitive closure of the relation defined by the axioms and inference rules below give the congruence relation among combinators possessing the same domain and codomain types:

- $(c_0 ; c_1) ; c_2 \equiv c_0 ; (c_1 ; c_2)$.
- $id ; c \equiv c \equiv c ; id$.
- $\langle c_0, c_1 \rangle ; p_0 \equiv c_0$.

- $\langle c_0, c_1\rangle; p_1 \equiv c_1$.
- $\langle c; p_0, c; p_1\rangle \equiv c$
- $c; ! \equiv !$.
- If $c_0 \equiv c_0'$ and $c_1 \equiv c_1'$ then $c_0; c_1 \equiv c_0'; c_1'$.
- If $c_0 = c_1$ in $\mathscr{E}$ then $c_0 \equiv c_1$.

When a finite product combinator theory is augmented by the declaration of a datatype it is necessary to add the required combinators (constructors/destructors and fold/unfold) satisfying the appropriate recursion diagrams as indicated in the introduction.

4.1. *Translating programs to combinators*

A translation, herein denoted $\mathscr{C}$, of closed abstracted terms, or programs, to combinators and a proof of its consistency is presented in this section. We first define below the translation of programs in the term logic. The notation

$$\mathscr{C}[\![v \mapsto t]\!]$$

represents the application of the translation $\mathscr{C}$ to the program $\{v \mapsto t\}$. The definition proceeds inductively on the construction of the abstracted term representing the program:

(i) $\mathscr{C}[\![v_\tau \mapsto ()]\!] = !_\tau$.

(ii) If x_τ is a variable then $\mathscr{C}[\![x_\tau \mapsto x_\tau]\!] = id_\tau$.

(iii) If v_0 and v_1 are variable bases then

$$\mathscr{C}[\![(v_0, v_1) \mapsto x]\!] = p_0; \mathscr{C}[\![v_0 \mapsto x]\!] \text{ if } x \in fvars(v_0))$$

and

$$\mathscr{C}[\![(v_0, v_1) \mapsto x]\!] = p_1; \mathscr{C}[\![v_1 \mapsto x]\!] \text{ if } x \in fvars(v_1).$$

(iv) $\mathscr{C}[\![v \mapsto \{v' \mapsto t\}(t')]\!] = \langle \mathscr{C}[\![v \mapsto t']\!], id\rangle; \mathscr{C}[\![(v', v) \mapsto t]\!]$.

(v) If f is a function symbol (including projections) then $\mathscr{C}[\![v \mapsto f(t)]\!] = \mathscr{C}[\![v \mapsto t]\!]; f$.

(vi) $\mathscr{C}[\![v \mapsto (t_0, t_1)]\!] = \langle \mathscr{C}[\![v \mapsto t_0]\!], \mathscr{C}[\![v \mapsto t_1]\!]\rangle$.

The rest of this section is devoted to showing the consistency of this translation. Appropriately, the generating set of given equations $\mathscr{E}$ in the target cartesian combinator theory should extend the $\equiv$-relation defined above and be exactly the translation images of those in E_0, the equations of the cartesian theory.

Whenever strong datatypes are declared, the translation requires an extension to the new terms generated by the addition of the associated constructors and factorizers. It is technically necessary to define the extension for only the constructors and the fold terms (for an initial datatype) or the destructors and the unfold terms (for a final datatype) since the remaining terms are expressible in terms of folds and unfolds. Yet it is instructive to see the development for the case and map terms as well and therefore they have been presented here with their specialized combinators. Shown below is the incremental extension to be used for adding a strong initial datatype L:

(i) $\mathscr{C}[\![v \mapsto c_i(t)]\!] = \mathscr{C}[\![v \mapsto t]\!]; c_i$

(ii)

$$\mathscr{C}\left[\!\!\left[v_X \mapsto \begin{Bmatrix} c_1 : v_1 \mapsto t_1 \\ \cdots \\ c_n : v_n \mapsto t_n \end{Bmatrix}(t) \right]\!\!\right]$$
$$= \langle \mathscr{C}[\![v_X \mapsto t]\!], id \rangle ; fold^L \{ \mathscr{C}[\![(v_1, v_X) \mapsto t_1]\!], \ldots, \mathscr{C}[\![(v_n, v_X) \mapsto t_n]\!] \}$$

(iii)

$$\mathscr{C}\left[\!\!\left[v_X \mapsto \begin{Bmatrix} c_1(v_1) \mapsto t_1 \\ \cdots \\ c_n(v_n) \mapsto t_n \end{Bmatrix}(t) \right]\!\!\right]$$
$$= \langle \mathscr{C}[\![v_X \mapsto t]\!], id \rangle ; case^L \{ \mathscr{C}[\![(v_1, v_X) \mapsto t_1]\!], \ldots, \mathscr{C}[\![(v_n, v_X) \mapsto t_n]\!] \}$$

(iv)

$$\mathscr{C}\left[\!\!\left[v_X \mapsto L \begin{bmatrix} w_1 \mapsto t_1 \\ \cdots \\ w_m \mapsto t_m \end{bmatrix}(t) \right]\!\!\right]$$
$$= \langle \mathscr{C}[\![v_X \mapsto t]\!], id \rangle ; map^L \{ \mathscr{C}[\![w_1, v_X) \mapsto t_1]\!]), \ldots, \mathscr{C}[\![(w_m, v_X) \mapsto t_m]\!] \}$$

Next are the translation rules to be added when adjoining a new strong final datatype R:

(i) $\mathscr{C}[\![v \mapsto d_i(t)]\!] = \mathscr{C}[\![v \mapsto t]\!]; d_i$

(ii)

$$\mathscr{C}\left[\!\!\left[v_X \mapsto \left\langle v_c \mapsto \begin{matrix} d_1 : t_1 \\ \cdots \\ d_n : t_n \end{matrix} \right\rangle(t) \right]\!\!\right]$$
$$= \langle \mathscr{C}[\![v_X \mapsto t]\!], id \rangle ; unfold^R \{ \mathscr{C}[\![(v_C, v_X) \mapsto t_1]\!], \ldots, \mathscr{C}[\![(v_C, v_X) \mapsto t_n]\!] \}$$

(iii)

$$\mathscr{C}\left[\!\!\left[v_X \mapsto \left\langle \begin{matrix} d_1 : t_1(t) \\ \cdots \\ d_n : t_n(t) \end{matrix} \right\rangle \right]\!\!\right]$$
$$= \langle \mathscr{C}[\![v_X \mapsto t]\!], id \rangle ; record^R \{ \mathscr{C}[\![(v_C, v_X) \mapsto t_1]\!], \ldots, \mathscr{C}[\![(v_C, v_X) \mapsto t_n]\!] \}$$

(iv)

$$\mathscr{C}\left[\!\!\left[v_X \mapsto R \begin{bmatrix} w_1 \mapsto t_1 \\ \cdots \\ w_m \mapsto t_m \end{bmatrix}(t) \right]\!\!\right]$$
$$= \langle \mathscr{C}[\![v_X \mapsto t]\!], id \rangle ; map^R \{ \mathscr{C}[\![w_1, v_X) \mapsto t_1]\!], \ldots, \mathscr{C}[\![(w_m, v_X) \mapsto t_m]\!] \}$$

Our aim is to prove the following.

Proposition 4.1. *For any theory built from a finite product theory augmented by a finite sequence of datatype declarations the translation $\mathscr{C}$ above is consistent.*

To facilitate the proof of this, which we will sketch, the following lemma is useful.

Lemma 4.2. *If $\{v \mapsto t\}$ is a closed abstraction, then*
(i) $\mathscr{C}[\![(v,v') \mapsto t]\!] = p_0 ; \mathscr{C}[\![v \mapsto t]\!]$
(ii) $\mathscr{C}[\![(v',v) \mapsto t]\!] = p_1 ; \mathscr{C}[\![v \mapsto t]\!]$.

Intuitively, this says that one can eliminate variables not used actively in an expression and this is a desirable practical optimization of the translations provided. The lemma is proven by structural induction on t. To illustrate the method we demonstrate the inductive step on a fold factorizer.

Suppose that (using abbreviated notation)

$$t = \{\!|c_i : v_i \mapsto t_i|\!\}(t_0)$$

then

$$\begin{aligned}
\mathscr{C}[\![(v,v') \mapsto \{\!|c_i : v_i \mapsto t_i|\!\}(t_0)]\!] &= \langle \mathscr{C}[\![(v,v') \mapsto t_0]\!], id\rangle ; fold^L\{\mathscr{C}[\![(e_i,(v,v')) \mapsto t_i]\!]\} \\
&= \langle p_0 ; \mathscr{C}[\![v \mapsto t_0]\!], id\rangle ; fold^L\{id \times p_0 ; \mathscr{C}[\![(e_i,v) \mapsto t_i]\!]\} \\
&= \langle p_0 ; \mathscr{C}[\![v \mapsto t_0]\!], id\rangle ;\ id \times p_0 ; fold^L\{\mathscr{C}[\![(e_i,v) \mapsto t_i]\!]\} \\
&= p_0 ; \langle \mathscr{C}[\![v \mapsto t_0]\!], id\rangle ; fold^L\{\mathscr{C}[\![(e_i,v) \mapsto t_i]\!]\} \\
&= p_0 ; \mathscr{C}[\![v \mapsto \{\!|c_i : e_i \mapsto t_i|\!\}(t_0)]\!] .
\end{aligned}$$

Notice that we have used the naturality of the strength and some projection manipulations.

Turning to the proof of the proposition it is necessary to prove the consistency of the inference rules of the term logic. This is done by checking each term logic inference rule in turn. In fact, all except the abstraction application (β-reduction) are now straightforward.

To show that $\{v \mapsto v\} = \{v' \mapsto v'\}$ it suffices to show that $\mathscr{C}[\![v \mapsto v]\!] = id$. This can be proven by a simple structural induction on the structure of the variable base:

basis:

$$\mathscr{C}[\![x \mapsto x]\!] = id$$

induction:

$$\begin{aligned}
\mathscr{C}[\![(v_0,v_1) \mapsto (v_0,v_1)]\!] &= \langle \mathscr{C}[\![(v_0,v_1) \mapsto v_0]\!], \mathscr{C}[\![(v_0,v_1) \mapsto v_1]\!]\rangle \\
&= \langle p_0 ; \mathscr{C}[\![v_0 \mapsto v_0]\!], p_1 ; \mathscr{C}[\![v_1,v_1]\!]\rangle \\
&= \langle p_0 ; id, p_1 ; id\rangle \\
&= id .
\end{aligned}$$

The unit rule is immediate.

To show the projection identities hold we have

$$\begin{aligned}\mathscr{C}[\![v \mapsto p_0(t_0,t_1)]\!] &= \mathscr{C}[\![v \mapsto (t_0,t_1)]\!]; p_0\\ &= \langle \mathscr{C}[\![v \mapsto t_0]\!], \mathscr{C}[\![v \mapsto t_1]\!]\rangle; p_0\\ &= \mathscr{C}[\![v \mapsto t_0]\!].\end{aligned}$$

For the congruence inference rule we note that when $\{v_0 \mapsto t_0\}$ and $\{v_1 \mapsto t_1\}$ are closed abstractions which are composable then we have

$$\begin{aligned}\mathscr{C}[\![v_0 \mapsto \{v_1 \mapsto t_1\}(t_0)]\!] &= \langle \mathscr{C}[\![v_0 \mapsto t_0]\!], id\rangle; \mathscr{C}[\![(v_1,v_0) \mapsto t_1]\!]\\ &= \langle \mathscr{C}[\![v_0 \mapsto t_0]\!], id\rangle; p_0; \mathscr{C}[\![v_1 \mapsto t_1]\!]\\ &= \mathscr{C}[\![v_0 \mapsto t_0]\!]; \mathscr{C}[\![v_1 \mapsto t_1]\!]\end{aligned}$$

from which the congruence inference follows easily.

It remains to show that $\mathscr{C}[\![v \mapsto \{v' \mapsto t'\}(t)]\!] = \mathscr{C}[\![v \mapsto \sigma_{v'=t}(t')]\!]$ and, of course, this is the heart of the proof. It is proven by a structural induction on t'. Again we sample this induction to give a flavor of the manipulations involved by showing the steps for an unfold factorizer.

Let $t' = (\!|v \mapsto d_i : t_i|\!)(t_0)$. Then

$$\begin{aligned}&\mathscr{C}[\![v \mapsto \sigma_{v'=t}((\!|v_0 \mapsto d_i : t_i|\!)(t_0))]\!]\\ &\quad= \mathscr{C}[\![v \mapsto (\!|v_0 \mapsto d_i : \sigma_{v'=t}(t_i)|\!)\sigma_{v'=t}(t_0)]\!]\\ &\quad= \langle \mathscr{C}[\![v \mapsto \sigma_{v'=t}(t_0)]\!], id\rangle; unfold_R\{\mathscr{C}[\![v_0, v) \mapsto \sigma_{v'=t}(t_i)]\!]\}\\ &\quad= \langle \mathscr{C}[\![v \mapsto \{v' \mapsto t_0\}(t)]\!], id\rangle; unfold_R\{\mathscr{C}[\![(v_0, v) \mapsto \{v' \mapsto t_i\}(t_i)]\!]\}\\ &\quad= \langle\langle \mathscr{C}[\![v \mapsto t]\!], id\rangle; \mathscr{C}[\![(v', v) \mapsto t_0]\!], id\rangle;\\ &\qquad unfold_R\{\langle \mathscr{C}[\![(v_i, v) \mapsto t]\!], id\rangle; \mathscr{C}[\![(v',(v_0, v)) \mapsto t_i]\!]\}\\ &\quad= \langle\langle \mathscr{C}[\![v \mapsto t]\!], id\rangle; \mathscr{C}[\![(v', v) \mapsto t_0]\!], id\rangle;\\ &\qquad unfold_R\{\langle p_1; \mathscr{C}[\![v \mapsto t]\!], id\rangle; \mathscr{C}[\![(v',(v_0, v)) \mapsto t_i]\!]\}\\ &\quad= \langle\langle \mathscr{C}[\![v \mapsto t]\!], id\rangle; \mathscr{C}[\![(v', v) \mapsto t_0]\!], id\rangle;\\ &\qquad unfold_R\{id \times \langle \mathscr{C}[\![v \mapsto t]\!], id\rangle; \mathscr{C}[\![(v_0,(v', v)) \mapsto t_i]\!]\}\\ &\quad= \langle\langle \mathscr{C}[\![v \mapsto t]\!], id\rangle; \mathscr{C}[\![(v', v) \mapsto t_0]\!], id\rangle; id \times \langle \mathscr{C}[\![v \mapsto t]\!], id\rangle;\\ &\qquad unfold_R\{\mathscr{C}[\![v_0,(v', v)) \mapsto t_i]\!]\}\\ &\qquad= \langle \mathscr{C}[\![v \mapsto t]\!], id\rangle; \mathscr{C}[\![(v', v) \mapsto t_0]\!], id\rangle; unfold_R\{\mathscr{C}[\![v_0,(v', v)) \mapsto t_i]\!]\}\\ &\qquad= \langle \mathscr{C}[\![v \mapsto t]\!], id\rangle; \mathscr{C}[\![(v', v) \mapsto (\!|v_0 \mapsto d_i : t_i|\!)(t_0)]\!]\\ &\qquad= \mathscr{C}[\![v \mapsto \{v' \mapsto (\!|v_0 \mapsto d_i : t_i|\!)\}(t_0)]\!].\end{aligned}$$

It remains only to check the consistency of the axioms introduced by the declared datatypes. For these it suffices to check the fold/unfold uniqueness and the action of the constructors and destructors. These rules translate directly into the recursion diagrams, however, and so the proof of consistency is complete.

4.2. *Translating from combinators to term logic*

In order to complete the proof of equivalence we may take one of the two routes. Either we can provide an inverse consistent translation or we can rely on the universal property of the combinator theory (as the free category generated by the given sequence of declarations). The first route involves lengthy, but essentially straightforward, structural inductions. It also provides an explicit translation which has practical ramifications. The second route is more conceptual but has the advantage of shedding a different light on the proof. For this reason while we actually follow the first method we discuss the second method below.

The second method starts with the observation collected from Section 2 that the closed abstractions of the term logic under composition from a category. Actually, not just any category: one with finite products and the declared datatypes. The combinator theory, by construction, is the free such category. Structure-preserving functors from the free category are totally determined by their action on the generators. Thus, if the translation to the term logic and the translation back to the combinators are structure preserving, and do not change the generators then the composed translation is the identity. To prove that the translations give an isomorphism of categories it then suffices to show that every closed abstraction of the term logic is (equivalent to) the translation of something in the combinator theory (note that the objects are unchanged in the translation).

Now the translation, almost by definition, preserves the generators and structure. Thus, the only remaining aspect would be to ensure that every closed abstraction is equivalent to a translated combinator expression!

Returning to the more direct first method: we start by presenting a translation in the opposite direction. The translation of a combinator f will be denoted $\mathscr{P}[\![f]\!]$. Two notations we shall use in this translation are the composition (";") and pairing (",") of closed abstractions:

(i) $\{v \mapsto t\}; \{v' \mapsto t'\} \equiv \{v \mapsto \sigma_{v'=t}(t')\}$

(ii) $\langle \{v \mapsto t_0\}, \{v' \mapsto t_1'\} \equiv \{v \mapsto (t_0, \sigma_{v'=v}(t_1))\}$.

In general, the translation of a combinator will require making choices of variable bases. As the choices do not affect the outcome we have not specified how they are made. The translation $\mathscr{P}$ is defined as follows:

(i) $\mathscr{P}[\![f;g]\!] = \mathscr{P}[\![f]\!]; \mathscr{P}[\![g]\!]$

(ii) $\mathscr{P}[\![\langle f;g\rangle]\!] = \langle \mathscr{P}[\![f]\!]; \mathscr{P}[\![g]\!]\rangle$

(iii) $\mathscr{P}[\![id_\tau]\!] = \{v_\tau \mapsto v_\tau\}$

(iv) $\mathscr{P}[\![!_\tau]\!] = \{v_\tau \mapsto ()\}$

(v) If v is a variable base for the domain of a primitive combinator, f, (including projections) then $\mathscr{P}[\![f]\!] = \{v \mapsto f(v)\}$.

The translation for the combinators for all three factorizers introduced by an inductive datatype declaration is presented next:

(i) $\mathscr{P}[\![c_i]\!] = \{v \mapsto c_i(v)\}$ where v is a variable base for the domain of c_i

(ii) $$\mathscr{P}[\![fold^L\{h_1,\ldots,h_n\}]\!] = \{(v_L, v_X) \mapsto \left\langle \begin{matrix} c_1 : v_1 \mapsto \mathscr{P}[\![h_1]\!](v_1, v_X) \\ \cdots \\ c_n : v_n \mapsto \mathscr{P}[\![h_n]\!](v_n, v_X) \end{matrix} \right\rangle (v_L)\}$$

(iii) $$\mathscr{P}[\![case^L\{h_1,\ldots,h_n\}]\!] = \{(v_L, v_X) \mapsto \left[\begin{matrix} c_1(v_1) \mapsto \mathscr{P}[\![h_1]\!](v_1, v_X) \\ \cdots \\ c_n(v_n) \mapsto \mathscr{P}[\![h_n]\!](v_n, v_X) \end{matrix} \right] (v_L)\}$$

(iv) $$\mathscr{P}[\![map^L\{f_1,\ldots,f_m\}]\!] = \{(v_L, v_X) \mapsto L \left[\begin{matrix} w_1 \mapsto \mathscr{P}[\![f_1]\!](w_1, v_X) \\ \cdots \\ w_m \mapsto \mathscr{P}[\![f_m]\!](w_m, v_X) \end{matrix} \right] (v_L)\}.$$

Similarly, the translation of the combinators introduced by a coinductive declaration is presented below:

(i) $\mathscr{P}[\![d_i]\!] = \{v \mapsto d_i(v)\}$ where v is a variable base for the domain of d_i

(ii) $$\mathscr{P}[\![unfold^R\{g_1,\ldots,g_n\}]\!] = \{(v'_C, v_X) \mapsto \left\langle v_C \mapsto \begin{matrix} d_1 : \mathscr{P}[\![g_1]\!](v_C, v_X) \\ \cdots \\ d_n : \mathscr{P}[\![g_n]\!](v_C, v_X) \end{matrix} \right\rangle (v'_C)\}$$

(iii) $$\mathscr{P}[\![record^R\{g_1,\ldots,g_n\}]\!] = \{(v'_C, v_X) \mapsto \left(\begin{matrix} d_1 : \mathscr{P}[\![g_1]\!](v'_C, v_X) \\ \cdots \\ d_n : \mathscr{P}[\![g_n]\!](v'_C, v_X) \end{matrix} \right)\}$$

(iv) $$\mathscr{P}[\![map^R\{f_1,\ldots,f_m\}]\!] = \{(v_R, v_X) \mapsto R \left(\begin{matrix} w_1 \mapsto \mathscr{P}[\![f_1]\!](w_1, v_X) \\ \cdots \\ w_m \mapsto \mathscr{P}[\![f_m]\!](w_m, v_X) \end{matrix} \right) (v_R)\}$$

Proposition 4.3. *For any finite product category augmented by a series of datatype declarations, $\mathscr{P}$ is a consistent translation of the combinator notation into term logic notation.*

To prove this we must go through the rules of the combinator theory. The usual rules of category theory, associativity of composition and the identity rules are easily verified. Furthermore, the recursion diagrams for a datatype translate almost directly into the term logic rules. To illustrate the technique we demonstrate the consistency of

the translation of two identities. For a fold factorizer reduction we have

$$
\begin{aligned}
&\mathscr{P}[\![c_i \times id_X ; fold^L\{h_1,\ldots,h_n\}]\!] \\
&\quad = \langle \mathscr{P}[\![p_0]\!] ; \mathscr{P}[\![c_i]\!], \mathscr{P}[\![p_1]\!]\rangle ; \mathscr{P}[\![fold^L\{h_1,\ldots,h_n\}]\!] \\
&\quad = \{(v_i, v_X) \mapsto (c_i(p_0(v_i, v_X)), p_1(v_i, v_X))\} ; \{(v_L, v_X) \mapsto \{\!|c_i : v_i \mapsto \mathscr{P}[\![h_i]\!](v_i, v_X)|\!\}(v_L)\} \\
&\quad = \{(v_i, v_X) \mapsto \{\!|c_i : v_i \mapsto \mathscr{P}[\![h_i]\!](v_i, v_X)|\!\}(c_i(v_i))\} \\
&\quad = \{(v_i, v_X) \mapsto \sigma_{v=E_i}\begin{bmatrix} w_A \mapsto w_A \\ w_L \mapsto \{\!|c_i : v_i \mapsto \mathscr{P}[\![h_i]\!](v_i, v_X)|\!\}(w_L)\end{bmatrix}(v_i)(\mathscr{P}[\![h_i]\!](v_i, v_X))\} \\
&\quad = \{(v_i, v_X) \mapsto (E_i\begin{bmatrix} w_A \mapsto \mathscr{P}[\![p_0]\!](w_A, v_X) \\ w_L \mapsto \mathscr{P}[\![fold^L\{h_1,\ldots,h_n\}]\!](w_L, v_X)\end{bmatrix}(v_i), (v_X)\} ; \\
&\qquad \{(v_i, v_X) \mapsto \mathscr{P}[\![h_i]\!](v_i, v_X)\} \\
&\quad = \langle \mathscr{P}[\![map^{E_i}\{p_0, fold^L\{h_1,\ldots,h_n\}\}]\!], \mathscr{P}[\![p_1]\!]\rangle ; \mathscr{P}[\![h_i]\!] \\
&\quad = \mathscr{P}[\![\langle map^{E_i}\{p_0, fold^L\{h_1,\ldots,h_n\}\}, p_1\rangle ; h_i]\!]
\end{aligned}
$$

The consistency for an R unfold factorizer reduction is demonstrated by

$$
\begin{aligned}
&\mathscr{P}[\![unfold^R\{g_1,\ldots,g_n\} ; d_i]\!] \\
&\quad = \mathscr{P}[\![unfold^R\{g_1,\ldots,g_n\}]\!] ; \mathscr{P}[\![d_i]\!] \\
&\quad = \{(v'_C, v_X) \mapsto \{\!|v_C \mapsto d_j : \mathscr{P}[\![g_j]\!](v_C, v_X)|\!\}(v'_C)\} ; \{v_R \mapsto d_i(v_R)\} \\
&\quad = \{(v'_C, v_X) \mapsto d_i(\{\!|v_C \mapsto d_j : \mathscr{P}[\![g_j]\!](v_C, v_X)|\!\}(v'_C))\} \\
&\quad = \{(v'_C, v_X) \mapsto E_i\begin{bmatrix} w_A \mapsto w_A \\ w_R \mapsto \{\!|v_C \mapsto d_j : \mathscr{P}[\![g_j]\!](v_C, v_X)|\!\}(w_C)\end{bmatrix}(\mathscr{P}[\![g_i]\!](v'_C, v_X)\} \\
&\quad = \{(v'_C, v_X) \mapsto (\mathscr{P}[\![g_i]\!](v'_C, v_X), v_X)\} ; \\
&\qquad \{(v_i, v_X) \mapsto E_i\begin{bmatrix} w_A \mapsto \mathscr{P}[\![p_0]\!](w_A, v_X) \\ w_R \mapsto \mathscr{P}[\![unfold^R\{g_1,\ldots,g_n\}]\!](w_C, v_X)\end{bmatrix}(v_i)\} \\
&\quad = \langle \mathscr{P}[\![g_i]\!], \mathscr{P}[\![p_1]\!]\rangle ; \mathscr{P}[\![map^{E_i}\{p_0, unfold^R\{g_1,\ldots,g_n\}\}]\!] \\
&\quad = \mathscr{P}[\![\langle g_i, p_1\rangle ; map^{E_i}\{p_0, unfold^R\{g_1,\ldots,g_n\}\}]\!]
\end{aligned}
$$

We can now state the main theorem of the paper.

Theorem 4.4. *$\mathscr{C}$ is an isomorphism, with inverse $\mathscr{P}$, between a finite product category augmented by datatype declarations presented in term logic notation and the category presented in combinator notation.*

Proof. Consider the translational composition $\mathscr{C} \circ \mathscr{P}$. The argument is a structural induction on the complexity of the term to be translated: we demonstrate the case of the fold factorizer. The inductive assumption is that on any smaller terms the

translation is the identity

$$\begin{aligned}
&\mathscr{C}[\![\mathscr{P}[\![fold^L\{h_1,\ldots,h_n\}]\!]]\!]\\
&\quad=\mathscr{C}[\![(v_L,v_X)\mapsto\{|c_i:v_i\mapsto\mathscr{P}[\![h_i]\!](v_i,v_X)|\}(v_L)]\!]\\
&\quad=fold^L\{\ldots,\mathscr{C}[\![(v_i,v_X)\mapsto\mathscr{P}[\![h_i]\!](v_i,v_X)]\!],\ldots\}\\
&\quad=fold^L\{\ldots,\mathscr{C}[\![\mathscr{P}[\![h_i]\!]]\!],\ldots\}\\
&\quad=fold^L\{\ldots,h_i,\ldots\}.
\end{aligned}$$

Now consider the composition $\mathscr{P}\circ\mathscr{C}$. Again the argument is a structural induction on the complexity of the terms to be translated. We demonstrate the fold factorizer case in the structural induction:

$$\begin{aligned}
&\mathscr{P}[\![\mathscr{C}[\![v_X\mapsto\{\!|c_i:v_i\mapsto t_i|\!\}(t)]\!]]\!]\\
&\quad=\mathscr{P}[\![\langle\mathscr{C}[\![v_X\mapsto t]\!],id\rangle;fold^L\{\ldots,\mathscr{C}[\![(v_i,v_X)\mapsto t_i]\!],\ldots\}]\!]\\
&\quad=\mathscr{P}[\![\langle\mathscr{C}[\![v_X\mapsto t]\!],id\rangle]\!];\mathscr{P}[\![fold^L\{\ldots,\mathscr{C}[\![(v_i,v_X)\mapsto t_i]\!],\ldots\}]\!]\\
&\quad=\langle\mathscr{P}[\![\mathscr{C}[\![v_X\mapsto t]\!]]\!],\mathscr{P}[\![id]\!]\rangle;\{(v_L,v_X)\mapsto\{\!|c_i:v_i\mapsto\mathscr{P}[\![\mathscr{C}[\![(v_i,v_X)\mapsto t_i]\!]]\!](v_i,v_X)|\!\}(v_L)\}\\
&\quad=\langle\{v_X\mapsto t\},\{v_X\mapsto v_X\}\rangle;\{(v_L,v_X)\mapsto\{\!|c_i:v_i\mapsto\{(v_i,v_X)\mapsto t_i\}(v_i,v_X)|\!\}(v_L)\}\\
&\quad=\{v_X\mapsto\{\!|c_i:v_i\mapsto\{(v_i,v_X)\mapsto t_i\}(v_i,v_X)|\!\}(t)\}\\
&\quad=\{v_X\mapsto\{\!|c_i:v_i\mapsto t_i|\!\}(t)\}.\qquad\square
\end{aligned}$$

5. Conclusions

A category can be not only a model of computation but also a medium.

The *charity* project at Calgary has verified that significant algorithms can be coded using a categorical programming language built atop this term logic. Our experience suggests that the more disciplined style of programming which is demanded by this language is easily learnt. Furthermore, it has often led the programmer to a clearer view of the structure of an algorithm.

Others, of course, have investigated calculi based in inductive and co-inductive datatypes. The interested reader should compare this work not only with Hagino's original work [5,6] but also with Malcolm's syntactic calculus [12] and Greiner's variant of the polymorphic lambda calculus [4]. While initial and final categorical datatypes underlie all these treatments, here we have provided a first-order useable programming language driven entirely by datatype declarations.

Categorical datatypes always satisfy standard models.

Of some interest is the role of parametricity within our setting and its connection to naturality. Hasegawa [7,8] has pointed out that parametricity is equivalent to the presence of categorical products and coproducts in models of polymorphic calculi. The setting we have discussed, of course, has true products and coproducts, and, furthermore, has no contravariant-type formation. Parametricity is therefore present for much simpler reasons.

Reynolds' abstraction theorem [16] asserts, under a precise definition of logical relations among sets, that related polymorphic functions produce related output values from related input values. This result has been specialized by Wadler [19] as a "free parametricity result" to give a more direct tool for proving program equivalences. Wadler's assertion that parametric theorems are "free" due to the apparent removal of any need for structural induction in their proofs has been challenged by Mairson [11]. The point of contention is Wadler's inclusion of a structurally inductive datatype (lists) into the definition of logical relations by which Wadler expresses his version of the abstraction theorem. In Mairson's mind, the question of whether an object in hand is actually a list for which a structural induction principle holds always requires answering.

In the setting described in this paper, inductive datatypes are always standard. Thus, Wadler's "theorems for free" apply without modification, indicating the promise of these developments for verification.

References

[1] J.R.B. Cockett, Distributive logic, Tech. Report CS-89-01, University of Tennessee, 1989.
[2] J.R.B. Cockett and T. Fukushima, About Charity, Research Report No. 92/480/18, Department of Computer Science, University of Calgary, 1992.
[3] J.R.B. Cockett and D. Spencer, Strong categorical datatypes I, in: R.A.G. Seely, ed., *International Meeting on Category Theory 1991, Canadian Mathematical Society Proceedings*, Vol. 13, AMS, Montreal (1992) 141–169.
[4] J. Greiner, Programming with inductive and co-inductive types, Tech. Report CMU-CS-92-109, Carnegie Mellon University, Pittsburgh, January 1992.
[5] T. Hagino, A Categorical programming language. Ph.D. Thesis, University of Edinburgh, 1987.
[6] T. Hagino, Codatatypes in ML, *J. Symbol. Comput.* **8** (1989) 629–650.
[7] R. Hasegawa, Categorical data types in parametric polymorphism, *Proc. 4th Asian Logic Conf.*, RIMS, 1990.
[8] R. Hasegawa, Parametricity of extensionally collapsed term models of polymorphism and their categorical properties, in: *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, Vol. 526 (Springer, Berlin, 1991) 495–512.
[9] J. Hughes, Why functional programming matters, *Comput. J.* **32** (2) (1989) 98–107.
[10] A. Kock, Strong functors and monoidal monads, *Arch. Math.* **23** (1972) 113–120.
[11] H.G. Mairson, Outline of a proof theory of parametricity, in: *Proc 6th Internat. Symp. on Functional Programming Languages and Computer Architecture* (1991) 313–327.
[12] G. Malcolm, Algebraic data types and program Transformation. Ph.D. Thesis, University of Groningen, 1990.
[13] E. Meijer, M. Fokkinga and R. Paterson, Functional programming with bananas, lenses, envelopes and barbed wire, in: *Proc. 6th Internat. Symp. on Functional Programming Languages and Computer Architecture*, 1991.

[14] L.C. Paulson, Co-induction and co-recursion in higher-order logic Tech. Report 304, University of Cambridge Computer Laboratory, July 1993.
[15] A.M. Pitts, A co-induction principle for recursively defined domains, Report 252, University of Cambridge Computer Laboratory, 1992.
[16] J.C. Reynolds, Types, abstraction and parametric polymorphism, in: R.E.A. Mason, ed., *Information Processing 83* (North-Holland, Amsterdam, 1983) 513–523.
[17] T. Sheard and L. Fegaras, A fold for all seasons, in: *Conf. Functional Programming Languages and Computer Architecture*, Copenhagen (1993) 233–242.
[18] A. Stoughton, Substitution revisited, *Theoret. Comput. Sci.* **59** (1988) 317–325.
[19] P. Wadler, Theorems for free! in: *Proc. 4th Internat. Symp. on Functional Programming Languages and Computer Architecture*, London, UK, September 1989.
[20] G.C. Wraith, *A Note on Categorical Datatypes*, Lecture Notes in Computer Science, Vol. 389 (Springer, Berlin, 1989) 118–127.