About Charity

Robin Cockett
Department of Computer Science
University of Calgary
Calgary, Alberta
CANADA
robin@cpsc.ucalgary.ca

Tom Fukushima
Department of Computer Science
University of Calgary
Calgary, Alberta
CANADA
fukushim@cpsc.ucalgary.ca

May 27, 1992

Abstract

Charity is a categorical programming language based on distributive categories (in the sense of Schanuel and Lawvere) with strong datatypes (in the sense of Hagino). Distributive categories come with a term logic which can express most standard programs; and they are fundamental to computer science because they permit proof by case analysis and, when strong datatypes are introduced, proof by structural induction.

Charity is functional and polymorphic in style, and is strongly normalizing. As a categorical programming language it provides a unique marriage of computer science and mathematical thought. The above aspects are particularly important for the production of verified programs as the naturality of morphisms gives us "theorems for free", termination proofs are not required, and mathemathical specifications can be used.

1 Introduction

Functional and logic programming languages have reduced the gap between theory and implementation by reducing the notational movement from mathematics to program, but some significant gaps persist. In particular, the use of general recursion, which produces "fixed points" in the semantics of these languages implies that they represent a peculiar version of mathematics.

The language charity starts from the basis that one can and should actually implement standard mathematics without trickery. Thus, we eliminate the uncomfortable transition

$$\textbf{discrete mathematics} \underset{?}{\overset{?}{\rightleftarrows}} \textbf{Program}$$

by basing the discrete mathematics in distributive categories and making the programming language charity which makes them the same pursuit.

Bob Walters, Steve Schanuel, Bill Lawvere, and Robin Cockett had all observed that of all the various sorts of categories available distributive categories seem to have exactly the right properties for doing computer science. Indeed, Robin Cockett had observed that the subject of discrete structures was really[1] about the properties of (the initial) distributive category with list arithmetic[Coc90].

Currently one of the most unsatisfactory aspects of software development is the movement from specification to program. There are various factors which exacerbate the achievement of a smooth transition. A fundamental problem is that, while one wants the specification to be in a language which is generally understandable, this is often in conflict with the requirement that it be rigorous and implementable.

The solution which seems to be emerging has been to link up formal mathematical methods more closely with programming. The idea being that if a specification is essentially a piece of mathematics it will be rigorous and, on the other hand, readable as we have been *taught* to read mathematics! Whatever one thinks of such a solution, it is clear that there is considerable value to the idea. There is, after all, a wealth of mathematical understanding out there which could be useful in developing specifications.

The point of charity is that it is an uncompromising implementation of constructive discrete mathematics. While it is certainly possible to implement constructive discrete mathematics in a functional language, it is not the case that the semantics of these languages embody constructive discrete mathematics. Certainly, there is a logic underlying functional (and imperative) programming, but it is not the logic which underpins or indeed is even compatible with the vast majority of our mathematical training. In fact, mathematically speaking it is decidedly peculiar!

In mathematics, functions are total, however, when programming over an imperative or functional language, one does have to be careful to avoid the production of non-terminating programs. As charity has a semantics which is compatible with this mathematical intuition one cannot write a non-terminating program. This is not to say that everything is automatically correct but rather that a subtle and all pervading source of incorrectness has been eliminated. For the program verifier this means that the hard proof of termination is no longer required.

The adoption of categorical or type theoretic languages like charity requires a change of the programming paradigm. In particular, general recursion, which is an absolutely fundamental tool in modern computer science thought, is not explicitly allowed in these paradigms. While charity is still under development, we already know that not using general recursion is possible theoretically, pragmatically, and practically. However, much work remains to bring this paradigm incisively to bear on practical applications.

2 Preliminaries

A recent trend in computer science is to see a category as being the proof theory of some known logic. Thus, cartesian closed categories are the proof theory of intuitionistic propositional logic, $*$–autonomous categories of the multiplicative fragment of linear logic. As computer scientists have often seen the usefulness of logic (implying that a similar statement concerning category theory is more tendentious) this provides a convenient justification of the use of categories. We may start therefore by asking rather cynically "for what logic are distributive categories the proof theory?" Fortunately, there is such a logic and it is the $(\wedge, \vee)$–fragment of propositional logic – or at least

[1] A computer scientist was asked what struck him most about category theorists. He replied that they always told him what subjects, which he had been studying for some time, were *really* about and this really annoyed him.

morally so!

We shall start by explaining what a distributive category is. This is somewhat complicated by the fact that there are several different shades of distributive categories with embedding theorems allowing one to slip relatively smoothly between them. We start with predistributive categories: a distributive category is a completion of this basic concept (in the same sort of sense that the complex numbers are a completion of the real numbers). We shall assume that everyone knows what a category is and what a product and coproduct are.

2.1 Notation

Roman font, lower-case italics and upper-case italics will be used to indicate items with predefined meanings, variables for maps and type variables respectively. If $f : X \longrightarrow Y$ and $g : Y \longrightarrow Z$ then the composition of the maps will be written, using the algebraic order, $f; g$.

The notation used for maps associated with the product and coproduct will be:

$$\begin{array}{rrl}
\text{identity} & 1: & X \longrightarrow X \\
\text{terminal map} & !: & Z \longrightarrow 1 \\
\text{product factorizer} & \langle x, y\rangle : & W \longrightarrow X \times Y \\
\text{first projection} & \mathrm{p}_0 : & X \times Y \longrightarrow X \\
\text{second projection} & \mathrm{p}_1 : & X \times Y \longrightarrow Y \\
\text{product symmetry} & \mathrm{c} = \langle \mathrm{p}_1, \mathrm{p}_0\rangle : & X \times Y \longrightarrow Y \times X \\
\text{product diagonal map} & \Delta = \langle 1, 1\rangle : & X \longrightarrow X \times X \\
\text{coproduct factorizer} & \langle v \mid w\rangle : & X + Y \longrightarrow W \\
\text{first coprojection} & \mathrm{b}_0 : & X \longrightarrow X + Y \\
\text{second coprojection} & \mathrm{b}_1 : & Y \longrightarrow X + Y
\end{array}$$

where

$$x : W \longrightarrow X,\ \ y : W \longrightarrow Y,\ \ v : X \longrightarrow W,\ \text{and}\ w : Y \longrightarrow W$$

and these satisfy the standard equations

$$\begin{array}{rcl}
z; ! & = & ! \\
\langle x, y\rangle; \mathrm{p}_0 & = & x \\
\langle x, y\rangle; \mathrm{p}_1 & = & y \\
\langle x; \mathrm{p}_0, x; \mathrm{p}_1\rangle & = & x \\
\mathrm{b}_0; \langle v \mid w\rangle & = & v \\
\mathrm{b}_1; \langle v \mid w\rangle & = & w \\
\langle \mathrm{b}_0; x \mid \mathrm{b}_1; x\rangle & = & x.
\end{array}$$

2.2 Predistributive categories

A **predistributive category** is a category with finite products and binary coproducts such that the map

$$\langle \mathrm{b}_0 \times 1 \mid \mathrm{b}_1 \times 1\rangle : (A \times X) + (B \times X) \longrightarrow (A + B) \times X$$

is invertible. When coproducts satisfy this we shall call them **sums**. In particular, given any maps:

$$f : A \times X \longrightarrow C \quad \text{and} \quad g : B \times X \longrightarrow C$$

there is a unique map

$$\mathrm{case}\{f, g\} : (A + B) \times X \longrightarrow C$$

defined by $\mathrm{case}\{f, g\} = \langle \mathrm{b}_0 \times 1 \mid \mathrm{b}_1 \times 1\rangle^{-1}; \langle f \mid g\rangle$. This can be thought of as the **if...then...else** program control construct: "if the value is in the first case do f else do g." It is precisely the ability of distributive categories to express control which makes them so applicable to computer science.

2.3 Proof theory

The way in which predistributive categories arise as the proof theory of the $(\wedge, \vee)$–fragment of propositional logic is as follows:

Axiom		
$A \vdash A$	identity	$A \xrightarrow{1} A$
Logical Rules		
$\frac{}{A \vdash \mathrm{true}}$	truth	$A \xrightarrow{!} 1$
$\frac{\Gamma \vdash B \quad \Gamma \vdash C}{\Gamma \vdash B \wedge C}$	$\wedge$–intro. and elim.	$\frac{\Gamma \xrightarrow{f} B \quad \Gamma \xrightarrow{g} C}{\Gamma \xrightarrow{\langle f,g\rangle} B \times C}$
$\frac{A,\Gamma \vdash C \quad B,\Gamma \vdash C}{A \vee B,\Gamma \vdash C}$	$\vee$–intro. and elim.	$\frac{A \times \Gamma \xrightarrow{f} C \quad B \times \Gamma \xrightarrow{g} C}{\mathrm{case}\{f,g\}:(A+B)\times\Gamma \longrightarrow C}$
Structural rules		
$\frac{\Gamma,A,B,\Lambda \vdash C}{\Gamma,B,A,\Lambda \vdash C}$	exchange	$\frac{\Gamma \times A \times B \times \Lambda \xrightarrow{h} C}{1 \times \mathrm{c} \times 1;h:\Gamma \times B \times A \times \Lambda \longrightarrow C}$
$\frac{\Gamma,A,A \vdash C}{\Gamma,A \vdash C}$	contraction	$\frac{\Gamma \times A \times A \xrightarrow{h} C}{1 \times \Delta;h:\Gamma \times A \longrightarrow C}$
$\frac{\Gamma \vdash B}{\Gamma,A \vdash B}$	weakening	$\frac{\Gamma \xrightarrow{h} B}{\mathrm{p}_0;h:\Gamma \times A \longrightarrow B}$
Cut Rule		
$\frac{\Gamma \vdash A \quad A,\Lambda \vdash B}{\Gamma,\Lambda \vdash B}$	cut	$\frac{\Gamma \xrightarrow{f} A \quad A \times \Lambda \xrightarrow{g} B}{f \times 1;g:\Gamma \times \Lambda \longrightarrow B}$

2.4 A remark on abstract data structures

Bob Walters particularly has exploited the presence of control in describing how the specification of data structures can be accomplished using the language of distributive categories. The specification of a stack in a distributive category may be expressed as follows:

```
Sorts:
      A,stack(A)
Operations:
      empty: 1 --> stack(A),
      push: A * stack(A) --> stack(A),
      pop: stack(A) --> 1 + A * stack(A)
Equations:
      (empty | push) ; pop = 1,
      pop ; (empty | push) = 1
```

This is a short sweet specification compared to others in the literature: it says that a stack on `A` is any object with elements which are either `empty` or of the form `push(A,X)`. The specification done for a general type `A` does not introduce any extraneous types, and *all* its models are what we intuitively expect stacks to be!

2.5 Distributive categories

A category is **distributive** (in the sense of Schanuel and Lawvere) in case it is finitely complete and has binary coproducts satisfying the following diagrammatic condition:

$$\begin{tikzcd}
A' \arrow[r, "f'"] \arrow[d, "a"'] \arrow[dr, phantom, "(1)"] & B' \arrow[d, "b"] \arrow[dr, phantom, "(2)"] & C' \arrow[l, "g'"'] \arrow[d, "c"] \\
A \arrow[r, "f"'] & B & C \arrow[l, "g"]
\end{tikzcd}$$

whenever the lower row is a coproduct, squares (1) and (2) are pullbacks if and only if the top row is a coproduct.

A distributive category is certainly predistributive. There are many examples of distributive categories: the category of sets, **Set**, and any topos, topological spaces, the dual of the category of commutative rings, finitely complete bicartesian closed categories (with inhabited non-empty types) and formal categories of datatypes. The last example being the motivation behind this work.

Observe that the coproduct embeddings in a distributive category must be monic by considering:

$$\begin{tikzcd}
A \arrow[r, "\mathrm{b}_0"] \arrow[d, "!"'] \arrow[dr, phantom, "\mathrm{pb}"] & A+B \arrow[d, "!+!"] \arrow[dr, phantom, "\mathrm{pb}"] & B \arrow[l, "\mathrm{b}_1"'] \arrow[d, "!"] \\
1 \arrow[r, "\mathrm{true}"'] & 1+1 & 1 \arrow[l, "\mathrm{false}"]
\end{tikzcd}$$

and using the fact that any map from the final object is monic so that its pullback along $!+!$ must be monic.

Considering, the pullback of the two embeddings into a coproduct we have an object 0 given by the left-hand pullback below:

$$\begin{tikzcd}
0 \arrow[r, "0"] \arrow[d, "!"'] \arrow[dr, phantom, "\mathrm{pb}"] & 1 \arrow[d, "\mathrm{false}"] \arrow[dr, phantom, "\mathrm{pb}"] & 1 \arrow[l, "1"'] \arrow[d, "1"] \\
1 \arrow[r, "\mathrm{true}"'] & 1+1 & 1 \arrow[l, "\mathrm{false}"]
\end{tikzcd}$$

This is an initial object. From the fact that the top row is a coproduct it follows that $0+1 \equiv 1$. However, this gives

$$\begin{tikzcd}
0 \arrow[r, "\mathrm{b}_0"] \arrow[d, "1"'] \arrow[dr, phantom, "\mathrm{pb}"] & 0+B \arrow[d, "1+!"] \arrow[dr, phantom, "\mathrm{pb}"] & B \arrow[l, "\mathrm{b}_1"'] \arrow[d, "!"] \\
0 \arrow[r, "\mathrm{b}_0"'] & 0+1 & 1 \arrow[l, "\mathrm{b}_1"]
\end{tikzcd}$$

in which the embeddings b_1 are isomorphisms.

Any two different maps $P \longrightarrow B$ will provide two different maps $P+B$ to B, whose second component is the identity. If $\mathrm{b}_1 : B \longrightarrow P+B$ is an isomorphism it follows that no two such maps can exist. If $\mathrm{b}_1 : B \longrightarrow P+B$ is an isomorphism for every B then there can be at most one map from P to B. In other words P is a preinitial object (at most one map to any other object).

The object 0 is therefore preinitial. However, there always is a map $\mathrm{b}_0; \mathrm{b}_1^{-1} : 0 \longrightarrow 0+B \longrightarrow B$ and so it is an initial object. Furthermore, it is a strict initial object (in the sense that any map with codomain 0 is an isomorphism) as any object P with a map to it is preinitial because $\mathrm{b}_1 : B \longrightarrow P+B$ is necessarily an isomorphism by pulling back over $0+1$.

This means that the coproduct in a distributive category is disjoint and universal. In fact, having disjoint universal coproducts implies the category is distributive.

As the defining diagrammatic condition of a distributive category is connected it will hold in any slice category. This means that if $\mathbf{X}$ is a distributive category then $\mathbf{X}/A$ is a distributive category.

We shall assume for the rest of this report that we are working in a distributive category.

3 Datatypes and term logic

The initial distributive category is the category of finite sets. While this is a very interesting category it is not sufficiently rich to model the data structures commonly used in computer science. Thus, we need to show how we may add datatypes in the sense of Hagino.

The purpose of this section is not only to introduce these datatypes but also to introduce the term logic which is used for their manipulation.

3.1 Number and list arithmetic

A distributive category has **number arithmetic** if it has a strong natural number object. A **strong natural number object** $(\mathcal{N}, 0, \mathrm{s})$ has an element $0 : 1 \longrightarrow \mathcal{N}$ and a successor map $\mathrm{s} : \mathcal{N} \longrightarrow \mathcal{N}$. Such that given any $f : X \longrightarrow C$ and endomorphism $g : C \longrightarrow C$ there is a unique map $\mathrm{fold}^{\mathcal{N}}\{f, g\}$ such that

$$
\begin{tikzcd}
X \arrow[r, "\langle !;0,1\rangle"] \arrow[dr, "f"'] & \mathcal{N}\times X \arrow[d, dotted, "\mathrm{fold}^{\mathcal{N}}\{f,g\}"] & \mathcal{N}\times X \arrow[l, "\mathrm{s}\times 1"'] \arrow[d, dotted, "\mathrm{fold}^{\mathcal{N}}\{f,g\}"] \\
& C & C \arrow[l, "g"]
\end{tikzcd}
$$

Morally speaking this is the wrong diagram it should really be

$$
\begin{tikzcd}
1\times X \arrow[r, "0\times 1"] \arrow[dr, "f"'] & \mathcal{N}\times X \arrow[d, dotted, "\mathrm{fold}^{\mathcal{N}}\{f,g\}"] & \mathcal{N}\times X \arrow[l, "\mathrm{s}\times 1"'] \arrow[d, dotted, "\langle \mathrm{fold}^{\mathcal{N}}\{f,g\},\mathrm{p}_1\rangle"] \\
& C & C\times X \arrow[l, "g"]
\end{tikzcd}
$$

However, by having X as a component of C in the first diagram we can obtain the effect of the second diagram.

It is well-known that a natural number object present in this strong form allows the representation of all the primitive recursive functions. Indeed the initial cartesian category with a strong natural number object *is* the formal theory of primitive recursive functions.

It is simple to see that a strong natural number object is preserved by slicing. In $\mathbf{X}/A$ the new natural number object is $[\mathcal{N}\times A \xrightarrow{\mathrm{p}_1} A]$. Furthermore, this is a strong natural number for $\mathbf{X}/A$.

A distributive category with list arithmetic, a **locos**, is the categorical setting for discrete structures. A distributive category has **list arithmetic** in case it has a strong list object for every A. A **strong list object** for A, $(\mathrm{list}(A), \mathrm{nil}, \mathrm{cons})$, has an element $\mathrm{nil} : 1 \longrightarrow \mathrm{list}(A)$ and an action $\mathrm{cons} : A\times \mathrm{list}(A) \longrightarrow \mathrm{list}(A)$ such that given any $f : 1\times X \longrightarrow C$ and action $g : (A\times C)\times X \longrightarrow C$ there is a unique map $\mathrm{fold}^{\mathrm{list}}\{f,g\}$ making the following diagram commute:

$$
\begin{tikzcd}
1\times X \arrow[r, "\mathrm{nil}\times 1"] \arrow[dr, "f"'] & \mathrm{list}(A)\times X \arrow[d, dotted, "\mathrm{fold}^{\mathrm{list}}\{f,g\}"] & (A\times \mathrm{list}(A))\times X \arrow[l, "\mathrm{cons}\times 1"'] \arrow[d, dotted, "\langle \theta^{\times}_{R}; 1\times \mathrm{fold}^{\mathrm{list}}\{f,g\},\mathrm{p}_1\rangle"] \\
& C & (A\times C)\times X \arrow[l, "g"]
\end{tikzcd}
$$

where $\theta^{\times}_{R} : (A\times \mathrm{list}(A))\times X \longrightarrow A\times(\mathrm{list}(A)\times X); ((x,y),z)\mapsto (x,(y,z))$ is the associative isomorphism. The notation $\theta^{\times}_{R}$ will be described in more detail later. A strong list object is an algebraically free monoid in the sense of Max Kelly and has associated with it a multiplication (given by append $=\mathrm{fold}^{\mathrm{list}}\{\mathrm{p}_1,\mathrm{p}_0;\mathrm{cons}\}$). The list object on the final object is a strong natural number object, thus, list arithmetic certainly implies number arithmetic. We would not expect the reverse to be true, so it is a little surprising to discover how nearly it is true. It suffices to have enough "A–infinite objects" to obtain the list arithmetic from number arithmetic[Coc91a].

If a distributive category has list arithmetic then all its slice categories will have list arithmetic. This is a non-trivial observation.

3.2 Datatypes

Both the strong natural number object and the list datatype are examples of the general notion of an initial datatype which was introduced by Tatsuya Hagino in his thesis[Hag87]. Gavin Wraith somewhat simplified the original system and declarations by assuming explicitly that the underlying setting was a bicartesian closed category[Wra89]. In this treatment we take another direction and assume that the underlying category is distributive and replace the closed requirement by the assumption that the datatypes are strong. We also include final datatypes, which are the dual of initial datatypes.

Initial datatypes are given by definitions of the form:

$$\begin{array}{rl} \textbf{data} & L(A) \longrightarrow S = \\ & c_1 : E_1(A,S) \longrightarrow S \\ & \vdots \\ | & c_n : E_n(A,S) \longrightarrow S. \end{array}$$

This says that the maps from the new type $L(A)$ to a type S are determined by maps to S from the types $E_1(A,S), ..., E_n(A,S)$. The type variable A is usually taken to range over a power (a finite product) of the given category. The object $L(A)$ comes equipped with canonical maps

$$c_j : E_j(A, L(A)) \longrightarrow L(A)$$

whose type is obtained by setting $S = L(A)$ in the declaration. These are called **constructors**. The functor L is called the **type construction** of the datatype. The datatype is given by the pair (L, c) of construction and constructors.

With respect to these maps the datatype satisfies the following universal property: given $g_i : E_i(A,Y) \times X \longrightarrow Y$ there is a unique map $\mathrm{fold}^L\{g_1, ..., g_n\}$, such that the following diagram commutes:

$$\begin{tikzcd}
E_i(A, L(A)) \times X \arrow[r, "c_i \times 1"'] \arrow[d, "\langle \theta_R^{E_i}, \mathrm{p}_1 \rangle"'] & L(A) \times X \arrow[dd, dotted, "\mathrm{fold}^L"] \\
E_i(A, L(A) \times X) \times X \arrow[d, dotted, "E_i\{1, \mathrm{fold}^L\} \times 1"'] & \\
E_i(A, Y) \times X \arrow[r, "g_i"] & Y
\end{tikzcd}$$

where $\theta_R^{E_i} = \theta^{E_i}; E_i\{p_0, 1\}$ and the subscripted R means to use the strength in the recursive argument only, that is, where $L(A)$ occurs. In the datatype (L, c), the construction L is necessarily a strong functor. It is well-known that if a category has products (or sums) and has initial datatypes then the constructors form a coproduct cone. For strong initial datatypes we have the stronger result: if (L, c) is an initial strong datatype then the constructors form a sum (where we use the term sum, as before, to indicate that the product distributes over this coproduct).

The strength of the type constructor L is defined uniquely by the following diagram:

$$
\begin{tikzcd}
E_i(A, L(A)) \times X \arrow[rr, "c_i \times 1"] \arrow[d, "\langle \theta_R^{E_i}, \mathrm{p}_1 \rangle"'] & & L(A) \times X \arrow[dd, dotted, "\theta^L"] \\
E_i(A, L(A) \times X) \times X \arrow[d, dotted, "E_i\{1, \theta^L\} \times 1"'] & & \\
E_i(A, L(A \times X)) \times X \arrow[r, "\theta_{\neg R}^{E_i}"'] & E_i(A \times X, L(A \times X)) \arrow[r, "c_i"'] & L(A \times X)
\end{tikzcd}
$$

The subscripted $\neg R$ on the strength means to use the strength in the non-recursive arguments only. Notice that

$$
\begin{array}{rcll}
c_i \times 1; \mathrm{map}^L\{f\} & = & c_i \times 1; \theta^L; L\{f\} & \\
 & = & \langle \theta_R^{E_i}, \mathrm{p}_1 \rangle; E_i\{1, \theta^L\} \times 1; \theta_{\neg R}^{E_i}; c_i; L\{f\} & \text{by definition} \\
 & = & \langle \theta_R^{E_i}, \mathrm{p}_1 \rangle; \theta_{\neg R}^{E_i}; E_i\{1, \theta^L\}; c_i; L\{f\} & \text{naturality of } \theta_{\neg R}^{E_i} \\
 & = & \theta^{E_i}; E_i\{1, \theta^L\}; c_i; L\{f\} & \text{corollary 3.4 below} \\
 & = & \theta^{E_i}; E_i\{1, \theta^L\}; E_i\{f, L\{f\}\}; c_i & \text{naturality of } c_i \\
 & = & \theta^{E_i}; E_i\{f, \theta^L; L\{f\}\}; c_i & \text{functorality of } E_i \\
 & = & \mathrm{map}^{E_i}\{f, \mathrm{map}^L\{f\}\}; c_i &
\end{array}
$$

where $\theta^{E_i}; E_i\{x, y\}$ and $\theta^L; L\{x\}$ are combined to give the map combinators $\mathrm{map}^{E_i}\{x, y\}$ and $\mathrm{map}^L\{x\}$ for the map operation which will be discussed again later. The equation also gives an optimized rewrite rule for the map combinator which will be used in the formulation of the abstract machine.

Final datatypes are given by definitions of the form:

$$
\begin{array}{lll}
\textbf{data} & & S \longrightarrow R(A) = \\
 & & d_1 : S \longrightarrow F_1(A, S) \\
 & & \vdots \\
 & | & d_n : S \longrightarrow F_n(A, S).
\end{array}
$$

Dual to the initial datatypes, this says that the maps from S to the new type $R(A)$ are determined by maps from S to the types $F_1(A, S)$, ..., $F_n(A, S)$. Again, the type variable A is taken to range over a power (a finite product) of the given category. The object $R(A)$ comes equipped with canonical maps

$$d_j : R(A) \longrightarrow F_j(A, R(A))$$

whose type is obtained by setting $S = R(A)$ in the declaration. These are called **destructors**. The functor R is called the **type constructor** of the datatype.

The datatype satisfies the following universal property: given $g_i : S \times X \longrightarrow F_i(A, S)$ there is a unique map $\mathrm{unfold}^R\{g_1, ..., g_n\}$, such that the following diagram commutes:

$$\begin{tikzcd}
S \times X \arrow[r, "\langle g_i, \mathrm{p}_1 \rangle"] \arrow[dd, dotted, "\mathrm{unfold}^R"'] & F_i(A, S) \times X \arrow[d, "\theta_R^{F_i}"] \\
 & F_i(A, S \times X) \arrow[d, dotted, "F_i(1, \mathrm{unfold}^R)"] \\
R(A) \arrow[r, "d_i"'] & F_i(A, R(A))
\end{tikzcd}$$

The strength of the type constructor R is defined uniquely by the following diagram:

$$\begin{tikzcd}
R(A) \times X \arrow[r, "d_i \times 1"] \arrow[dd, dotted, "\theta^R"'] & F_i(A, R(A)) \times X \arrow[r, "\langle \theta_{\neg R}^{F_i}, \mathrm{p}_1 \rangle"] & F_i(A \times X, R(A)) \times X \arrow[d, "\theta_R^{F_i}"] \\
 & & F_i(A \times X, R(A) \times X) \arrow[d, dotted, "F_i\{1, \theta^R\}"] \\
R(A \times X) \arrow[rr, "d_i"'] & & F_i(A \times X, R(A \times X))
\end{tikzcd}$$

Notice that

$$\begin{array}{rcll}
\mathrm{map}^R\{h\}; d_i & = & \theta^R; R\{h\}; d_i & \\
 & = & d_i \times 1; \langle \theta_{\neg R}^{F_i}, \mathrm{p}_1 \rangle; \theta_R^{F_i}; F_i\{1, \theta^R\}; F_i\{h, R\{h\}\} & \text{by definition} \\
 & = & d_i \times 1; \theta^{F_i}; F_i\{1, \theta^R\}; F_i\{h, R\{h\}\} & \text{corollary 3.4 below} \\
 & = & d_i \times 1; \theta^{F_i}; F_i\{h, \theta^R; R\{h\}\} & \text{functorality of } F_i \\
 & = & d_i \times 1; \mathrm{map}^{F_i}\{h, \mathrm{map}^R\{h\}\}. &
\end{array}$$

As with the initial datatypes, this identity will be used to give an efficient rewriting rule for the abstract machine.

The achieve the above equalities we need the following results.

Definition 3.1 *A bifunctor* $F : \mathbf{A} \times \mathbf{B} \longrightarrow \mathbf{C}$ *is strong if there is a natural transformation* $\theta^F_{A,B,X} : F(A,B) \times X \longrightarrow F(A \times X, B \times X)$.

Definition 3.2 *A bifunctor* $F : \mathbf{A} \times \mathbf{B} \longrightarrow \mathbf{C}$ *is bistrong if* $F(\_, B)$ *and* $F(A, \_)$ *are strong.*

Proposition 3.3 *A bifunctor is strong if and only if it is bistrong and*

$$\begin{tikzcd}
(F(A,B) \times X) \times Y \arrow[rr, "s"] \arrow[dr, "\theta^{F(A,_)}_{B,X} \times 1"'] & & (F(A,B) \times Y) \times X \arrow[d, "\theta^{F(,B)}_{A,Y} \times 1"] \\
& F(A, B \times X) \times Y \arrow[dr, "\theta^{F(_,B \times X)}_{A,Y}"'] & F(A \times Y, B) \times X \arrow[d, "\theta^{F(A \times Y,_)}_{B,X}"] \\
& & F(A \times Y, B \times X)
\end{tikzcd}$$

commutes, where $s = \langle\langle \mathrm{p}_0; \mathrm{p}_0, \mathrm{p}_1\rangle; \mathrm{p}_0; \mathrm{p}_1\rangle$.

Corollary 3.4 $\langle \theta^{E_i}_R, \mathrm{p}_1\rangle; \theta^{E_i}_{\neg R} = \theta^{E_i}$ *and* $\langle \theta^{E_i}_{\neg R}, \mathrm{p}_1\rangle; \theta^{E_i}_R = \theta^{E_i}$.

Proof. $\theta^{E_i}_{\neg R}$ and $\theta^{E_i}_R$ strengthen the first and second components of the bifunctor $E_i : \mathbf{A} \times \mathbf{C} \longrightarrow \mathbf{C}$ and so are $\theta^{E_i(\_,C)}$ and $\theta^{E_i(A,\_)}$ respectively. Thus,

$$\begin{array}{ll}
\multicolumn{2}{l}{\langle \theta^{E_i}_R, \mathrm{p}_1\rangle; \theta^{E_i}_{\neg R}} \\
= & \langle \theta^{E_i(A,_)}_{C,X}, \mathrm{p}_1\rangle; \theta^{E_i(_,C \times X)}_{A,X} \\
= & 1 \times \Delta; a; \theta^{E_i,(A,_)}_{B,X} \times 1; \theta^{E_i(_,C \times X)}_{A,X} \\
= & \theta^{E_i}.
\end{array}$$

Similarly, we can prove that $\langle \theta^{E_i}_{\neg R}, \mathrm{p}_1\rangle; \theta^{E_i}_R = \theta^{E_i}$.

$\square$

3.3 Term logic

It is sometimes convenient to describe the maps of a distributive category using a term logic. This removes the necessity to specify projections and can provide more natural looking proofs and programs. The term logic used here is the term logic of cartesian categories augmented with an appropriate notation for the sum and any other datatypes which have been declared. The notation used for maps from the sum is:

$$\mathrm{case}\{f,g\} : X \times (A+B) \longrightarrow C; (x,z) \mapsto \left\{ \begin{array}{ccc} \mathrm{b}_0(z_0) & \mapsto & f(x,z_0) \\ \mathrm{b}_1(z_1) & \mapsto & g(x,z_1) \end{array} \right\} (z).$$

In the combinator (or categorical notation) and term logic we shall use braces (which we shall often omit together with their contents when they can be inferred from the context) to enclose the arguments of a map whose definition is dependent on other maps (such as the arguments $\{f,g\}$ in $\mathrm{case}\{f,g\}$ above).

For the number arithmetic of the setting we shall sometimes use the special term notation

$$\mathrm{n}\{f,g\} : \mathcal{N} \times X \longrightarrow X; (n,x) \mapsto g^n(f(x)).$$

This notation suggests that the following equations should hold and they do:

$$\begin{array}{rcl} f^0(x) & = & x \\ f^n(f(x)) & = & f^{n+1}(x) \\ f^n(f^m(x)) & = & f^{n+m}(x) \\ (f^n)^m(x) & = & f^{n \cdot m}(x). \end{array}$$

For the initial datatype declared above there is a sum which is determined by its constructors. We may determine a map by its definition on each constructor. This gives the **case expression** of the charity programming language:

$$\mathrm{case}^L\{f_1,...,f_n\} : L(A) \times X \longrightarrow Y; (z,x) \mapsto \left\{ \begin{array}{ccc} c_1(z_1) & \mapsto & f_1(z_1,x) \\ & \vdots & \\ c_n(z_n) & \mapsto & f_n(z_n,x) \end{array} \right\} (z)$$

where $f_i : E_i(A, L(A)) \longrightarrow Y$.

The **fold operation** is written as

$$\mathrm{fold}^L\{g_1,...,g_n\} : L(A) \times X \longrightarrow C; (z,x) \mapsto \left\{\!\!\left\{ \begin{array}{ccc} c_1 : z_1 & \mapsto & g_1(z_1,x) \\ & \vdots & \\ c_n : z_n & \mapsto & g_n(z_n,x) \end{array} \right\}\!\!\right\} (z)$$

where $g_i : E_i(A,C) \times X \longrightarrow C$.

The **map expression** will be written as

$$\mathrm{map}^G\{h_1,...,h_m\} : G(A_1,...,A_m) \times X \longrightarrow G(B_1,...,B_m); (z,x) \longrightarrow G \left\{ \begin{array}{ccc} y_1 & \mapsto & h_1(y_1,x) \\ & \vdots & \\ y_m & \mapsto & h_m(y_m,x) \end{array} \right\} (z)$$

where $h_i : A_i \times X \longrightarrow B_i$.

For the final datatype declared above there is a product which is determined by its destructors. We may determine a map by its definition on each constructor. The **record expression** is written as:

$$\mathrm{record}^R\{f_1, ..., f_n\} : X \longrightarrow R(A); (x) \mapsto \left(\begin{array}{c} d_1 : f_1(x) \\ \vdots \\ d_m : f_n(x) \end{array} \right)$$

where $f_i : X \longrightarrow F_i(A, R(A))$.

The **unfold expression** is written as

$$\mathrm{unfold}^R\{g_1, ..., g_n\} : C \times X \longrightarrow R(A); (z, x) \mapsto \left(\!\!\left| z \mapsto \begin{array}{cc} d_1 : & g_1(z, x) \\ \vdots & \\ d_n : & g_n(z, x) \end{array} \right|\!\!\right) (z)$$

where $g_i : C \times X \longrightarrow F_i(A, C)$.

The **map expression** is the same as for the initial datatypes.

Note that although our notation for the term logic resembles the notation used by the squiggol community somewhat, they have different meanings. In [EMP91] the fold and unfold expressions are given by the banana and lens while our notation uses bananas for the unfold expression and "macaroons" for the fold expression. The reason for this is that right datatypes are product types and since we enclose pairs in parentheses, other operations on right data should also use the same syntax. The braces denote operations which operate mainly on left data. The map expression which operates on both left and right data uses braces.

4 The term logic in use

Rather than formally developing the term logic we shall concentrate on illustrating how it may be used. To this end we shall develop some simple charity programs using the term logic.

4.1 Boolean datatype

The simplest and most basic datatype is an object consisting of two copies of the final object 1. This is the "recognizable subobject classifier" and introduces a simple propositional logic into the setting.

$$\begin{array}{rl} \mathbf{data} & \mathrm{Bool} \longrightarrow X = \\ & \mathrm{true} : 1 \longrightarrow X \\ | & \mathrm{false} : 1 \longrightarrow X \end{array}$$

The expressions corresponding to "and," "or," and "not" are:

$$\wedge : \mathrm{Bool} \times \mathrm{Bool} \longrightarrow \mathrm{Bool}; (x_0, x_1) \mapsto \left\{ \begin{array}{ccc} \mathrm{true} & \mapsto & x_1 \\ \mathrm{false} & \mapsto & \mathrm{false} \end{array} \right\} (x_0)$$

$$\vee : \mathrm{Bool} \times \mathrm{Bool} \longrightarrow \mathrm{Bool}; (x_0, x_1) \mapsto \left\{ \begin{array}{ccc} \mathrm{true} & \mapsto & \mathrm{true} \\ \mathrm{false} & \mapsto & x_1 \end{array} \right\} (x_0)$$

$$\neg : \mathrm{Bool} \longrightarrow \mathrm{Bool}; x \mapsto \left\{ \begin{array}{ccc} \mathrm{true} & \mapsto & \mathrm{false} \\ \mathrm{false} & \mapsto & \mathrm{true} \end{array} \right\} (x)$$

Notice that all of these maps are given by the case expression. For this simple declaration this is the only expression of any significance. In fact, the fold operation and case expression coincide and the map expression has no arguments and so is the projection. It is easy although tedious to show that $\wedge$, $\vee$, and $\neg$ give Boolean algebra operations on Bool. The maps from any type to Bool are then the propositions of that type and induce a simple logic for the setting.

From the point of view of programming the most recognizable construct which may be produced when the Boolean datatype is present is the conditional:

$$\mathrm{If}\{p, f, g\} : Y \times (X_0 \times X_1) \longrightarrow Z; (y, (x_0, x_1)) \mapsto \left\{ \begin{array}{ccc} \mathrm{true} & \mapsto & f(x_0) \\ \mathrm{false} & \mapsto & g(x_1) \end{array} \right\} (p(y))$$

This says "if p then f else g." It takes as parameters a proposition $p : Y \longrightarrow \mathrm{Bool}$ and two functions $f : X_0 \longrightarrow Z$ and $g : X_1 \longrightarrow Z$.

4.2 Number Arithmetic

Probably the next most basic datatype is the natural numbers:

$$\begin{array}{lll} \mathbf{data} & \mathcal{N} \longrightarrow X = & \\ & \mathrm{zero} : 1 \longrightarrow X & \\ | & \mathrm{succ} : X \longrightarrow X & \end{array}$$

4.2.1 The case expression

Suppose we wish to express the map from $\mathcal{N}$ to $\mathcal{N}$ which is zero at zero and one elsewhere we would write:

$$\delta : \mathcal{N} \longrightarrow \mathcal{N}; n \mapsto \left\{ \begin{array}{ccc} \mathrm{zero} & \mapsto & \mathrm{zero} \\ \mathrm{succ}(\_) & \mapsto & \mathrm{succ}(\mathrm{zero}) \end{array} \right\} (n).$$

The meaning of this expression is literally "if n is zero the answer is zero else if n is the successor of some 'don't care' value set the answer to the successor of zero." Thus, the case expression does pattern matching on the structure of the term.

Here is another example of the use of the case expression in which we compute the predecessor of a number:

$$\mathrm{pred} : \mathcal{N} \longrightarrow \mathcal{N}; n \mapsto \left\{ \begin{array}{ccc} \mathrm{zero} & \mapsto & \mathrm{zero} \\ \mathrm{succ}(n') & \mapsto & n' \end{array} \right\} (n).$$

In this case again we output zero if n is zero, but this time on matching n and $\mathrm{succ}(n')$ we output n' the value obtained through the matching.

Here is a slightly more complex example which illustrates how one can use variables available from outside the case factorization. We shall assume that we have defined $x \dot{-} y$ (monus). Then note that one way of discovering whether $x > y$ is by determining whether $y \dot{-} x$ is zero or not. Thus, to determine the maximum of two numbers we have the following expression:

$$\max(n, m) : \mathcal{N} \times \mathcal{N} \longrightarrow \mathcal{N}; (n, m) \mapsto \left\{ \begin{array}{ccc} \mathrm{zero} & \mapsto & n \\ \mathrm{succ}(\_) & \mapsto & m \end{array} \right\} (m \dot{-} n).$$

Similarly, the expression for the minimum of two numbers is:

$$\min(n,m) : \mathcal{N} \times \mathcal{N} \longrightarrow \mathcal{N}; (n,m) \mapsto \left\{ \begin{array}{rcl} \text{zero} & \mapsto & m \\ \text{succ}(\_) & \mapsto & n \end{array} \right\} (m \dot{-} n).$$

However, this begs the question of how one expresses $\dot{-}$; for this we need the fold operation.

4.2.2 The fold operation

The fold operation of a datatype is the fundamental map associated with the datatype. Both the case expression and the map expression can be obtained from it. However, this does not mean that it is not worth having these last two present independently. They both have computational and expressive value. We have already seen that the case expression is quite useful expressively; to see why the map expression is useful we will have to wait until we discuss the list arithmetic. In the meantime let us consider the fold itself.

To express the function which adds two numbers we have two alternatives:

$$+ : \mathcal{N} \times \mathcal{N} \longrightarrow \mathcal{N}; (n,m) \mapsto \text{succ}^n(m)$$

or using the fold operation notation above:

$$+ : \mathcal{N} \times \mathcal{N} \longrightarrow \mathcal{N}; (n,m) \mapsto \left\{\!\!\left| \begin{array}{lcl} \text{zero} : () & \mapsto & m \\ \text{succ} : n' & \mapsto & \text{succ}(n') \end{array} \right|\!\!\right\} (n).$$

where we regard the second (succ) phrase of the operation to be a map which is iterated n times with starting conditions given by the first (zero) phrase. Clearly the first notation is much more suggestive, however, it does not generalize to arbitrary datatypes which the second notation, of course, does. For this reason we shall stress the latter syntax as this will lead into its use for more general datatypes later.

There is another way of viewing the meaning of the latter expression which is very suggestive. A natural number may be regarded as being a code in the sense that 2 means "take zero, apply succ, apply succ" the expression in the curly brackets indicates how this code is to be modified. Thus, on the instruction "take zero" one instead takes m and on the instruction "apply succ" one instead performs the second phrase, in this case (confusingly) to apply succ. The point being as the starting point is different the expression so interpreted computes $n + m$.

The expression for $n \dot{-} m$ is as follows

$$\dot{-} : \mathcal{N} \times \mathcal{N} \longrightarrow \mathcal{N}; (n,m) \mapsto \left\{\!\!\left| \begin{array}{lcl} \text{zero} : () & \mapsto & n \\ \text{succ} : n' & \mapsto & \text{pred}(n') \end{array} \right|\!\!\right\} (m).$$

In this case, regarding m as the code on the instruction "take zero" one instead takes n and on the instruction "apply succ" one instead applies pred.

The expression for multiplication is of interest as it uses strength in the recursive (succ) phrase; this would not be possible if we had used the first diagram in section 3.1 to give the fold operation:

$$\cdot : \mathcal{N} \times \mathcal{N} \longrightarrow \mathcal{N}; (n,m) \mapsto \left\{\!\!\left| \begin{array}{lcl} \text{zero} : () & \mapsto & \text{zero} \\ \text{succ} : n' & \mapsto & n' + n \end{array} \right|\!\!\right\} (m).$$

This may be regarded as saying add n m times to zero. It is interesting to express multiplication in the other syntax:

$$x \cdot y = \mathrm{p}_1(\{(n,m) \mapsto (n, m+n)\}^y(x, \mathrm{zero}))$$

which is not so nice!

Suppose we wish to create the proposition which tests whether a number is even. The way to do this is to use $\neg$:

$$\mathrm{even} : \mathcal{N} \longrightarrow \mathrm{Bool}; n \mapsto \left\{\!\!\left|\begin{array}{lcl} \mathrm{zero} : () & \mapsto & \mathrm{true} \\ \mathrm{succ} : x & \mapsto & \neg(x) \end{array}\right|\!\!\right\}(n),$$

which demonstrates how "regular" propositions on the natural numbers can arise. It is worth noting that eventually every map on a finite number of states either becomes fixed or cycles. This makes the regular propositions on the natural numbers eventually equivalent to a proposition of the form "p divides n."

The reader should try to write down the expression of other common arithmetic expressions in this syntax: for example the exponential and truncated halving (harder). As a last example consider the problem of writing factorial n:

$$\mathrm{fac} : \mathcal{N} \longrightarrow \mathcal{N}; n \mapsto \mathrm{p}_0(\left\{\!\!\left|\begin{array}{rcl} \mathrm{zero} : () & \mapsto & (\mathrm{succ}(\mathrm{zero}), \mathrm{succ}(\mathrm{zero})) \\ \mathrm{succ} : (n', m) & \mapsto & (n' \cdot m, \mathrm{succ}(m)) \end{array}\right|\!\!\right\}(n)).$$

Notice that here we have had to resort to a state which is a pair of values in order to express the factorial. This is needed as we need to have both the number we should multiply by and the factorial so far and *both are changing*.

Burstall gives an alternative syntax for the fold expression in [Bur87] which exploits the use of a cartesian closed setting.

4.3 List arithmetic

The list datatype has the following definition:

$$\begin{array}{lll} \textbf{data} & & \mathrm{list}(A) \longrightarrow X = \\ & & \mathrm{nil} : 1 \longrightarrow X \\ & | & \mathrm{cons} : A \times X \longrightarrow X \end{array}$$

in which nil is the empty list and cons is the operation which adds a new element to the front of the list.

4.3.1 The case expression

A map of classical interest is the following:

$$\mathrm{tail} : \mathrm{list}(A) \longrightarrow \mathrm{list}(A); z \mapsto \left\{\begin{array}{rcl} \mathrm{nil} & \mapsto & \mathrm{nil} \\ \mathrm{cons}(\_, z') & \mapsto & z' \end{array}\right\}(z).$$

What this does is to throw away the head of the list should there be one. Compare this to the map pred. Bearing in mind that list(1) is a natural number object we see that tail is the "generalization" of pred. Notice also that tail is not the pop of a stack although a simple way of implementing a stack is as a list.

A more complex use of the case expression is as follows: given a list of numbers return the sum of the first two numbers in the list. If the list does not contain two numbers return zero. This map is given by the following expression:

$$f : \mathrm{list}(\mathcal{N}) \longrightarrow \mathcal{N}; z \mapsto \left\{ \begin{array}{rcl} \mathrm{nil} & \mapsto & \mathrm{zero} \\ \mathrm{cons}(n_0, z') & \mapsto & \left\{ \begin{array}{rcl} \mathrm{nil} & \mapsto & \mathrm{zero} \\ \mathrm{cons}(n_1, \_) & \mapsto & n_0 + n_1 \end{array} \right\}(z') \end{array} \right\}(z)$$

which shows how the case expressions can be nested.

The simple pattern matching available for the case expression required that the case statements be nested in the above example. The ability to do more complex pattern matching is possible, but is not available yet.

4.3.2 The fold operation

The operation of appending two lists is given by the following fold operation:

$$\mathrm{append} : \mathrm{list}(A) \times \mathrm{list}(A) \longrightarrow \mathrm{list}(A); (z_1, z_2) \mapsto \left\{\!\!\left| \begin{array}{rcl} \mathrm{nil} : () & \mapsto & z_2 \\ \mathrm{cons} : (x, y) & \mapsto & \mathrm{cons}(x, y) \end{array} \right|\!\!\right\}(z_1).$$

In the same way that tail generalizes pred, the operation of appending two lists generalizes addition. Appending is confusing in exactly the same way as addition is confusing, as the second phrase in the fold operation uses the map cons. A useful way to regard this expression is to think of the list z_1 as being the code "take the empty list, push a_n onto the list, push a_{n-1} onto the list, ... , push a_1 onto the list." The two phrases of the fold operation then modify this code so that instead of starting with the empty list one starts with the list z_2 and then replacing each of the "push a_i onto the list"s with (the same) "push a_i onto the list."

A common thing one wants to do with lists is to reverse them. We shall show how to do this two different ways. First we shall do the naive reverse:

$$\mathrm{reverse} : \mathrm{list}(A) \longrightarrow \mathrm{list}(A); z \mapsto \left\{\!\!\left| \begin{array}{rcl} \mathrm{nil} : () & \mapsto & \mathrm{nil} \\ \mathrm{cons} : (a, y) & \mapsto & \mathrm{append}(y, \mathrm{cons}(a, \mathrm{nil})) \end{array} \right|\!\!\right\}(z).$$

It is well-known that this is inefficient as one repeatedly calls an append inside the second phrase of the fold operation acting on z (order n^2). Much more efficient is the following formulation of reverse (order n):

$$\begin{array}{l} \mathrm{reverse}' : \mathrm{list}(A) \longrightarrow \mathrm{list}(A); \\ z \;\mapsto\; \mathrm{p}_0(\left\{\!\!\left| \begin{array}{rcl} \mathrm{nil} : () & \mapsto & (\mathrm{nil}, z) \\ \mathrm{cons} : (\_, (y_0, y_1)) & \mapsto & \left\{ \begin{array}{rcl} \mathrm{nil} & \mapsto & (y_0, y_1) \\ \mathrm{cons}(a, y_1') & \mapsto & (\mathrm{cons}(a, y_0), y_1') \end{array} \right\}(y_1) \end{array} \right|\!\!\right\}(z)). \end{array}$$

Now we are claiming that they are the same maps, however we should really be able to prove this! We shall return to this in the next section.

Suppose that one has a list of lists, often one wants to flatten the list by appending all the inner lists together. This operation can be defined by:

$$\text{flatten} : \text{list}(\text{list}(A)) \longrightarrow \text{list}(A); z \mapsto \left\{\!\!\left|\begin{array}{rcl} \text{nil} : () & \mapsto & \text{nil} \\ \text{cons} : (z_0, z_1) & \mapsto & \text{append}(z_0, z_1) \end{array}\right|\!\!\right\}(z).$$

For example this has the effect of taking

$$[[a_1, a_2], [], [b_1, b_2]] \mapsto [a_1, a_2, b_1, b_2].$$

This is the generalization of the map which sums the entries of a list of natural numbers. It is also the "multiplication" of the list monad.

Perhaps the most common thing one wants to do with a list is to order the contents in some manner; this can be done with the following naive insertion sort. We will suppose that we are given a predicate $p : X \times X \longrightarrow \text{Bool}$ which determines the order (true if $x > y$): this will be used as a parameter in the map. To create the map which orders a list of X's it is first necessary to have a map which inserts an element into its proper place in an ordered list:

$$\begin{array}{l} \quad\text{insert}\{p\} : X \times \text{list}(X) \longrightarrow \text{list}(X); \\ (x, z) \;\; \mapsto \;\; \text{cons}(\left\{\!\!\left|\begin{array}{rcl} \text{nil} : () & \mapsto & (x, \text{nil}) \\ \text{cons} : (x_0, (x_1, z_1)) & \mapsto & \left\{\begin{array}{rcl} \text{true} & \mapsto & (x_1, \text{cons}(x_0, z_1)) \\ \text{false} & \mapsto & (x_0, \text{cons}(x_1, z_1)) \end{array}\right\}(p(x_0, x_1)) \end{array}\right|\!\!\right\}(z)). \end{array}$$

Notice that the map insert has the parameter p which allows this map to be general for all types with an order (in fact any proposition p will do). Also note that pushdown is not as efficient as it might be as it continues to do unnecessary comparisons after the inserted element has been placed in the list; this can be remedied by using a flag in the state to signal when the element has been inserted. The elements of the input list are then inserted, an element at a time, into the initially empty ordered list:

$$\text{insort}\{p\} : \text{list}(X) \longrightarrow \text{list}(X); z \mapsto \left\{\!\!\left|\begin{array}{rcl} \text{nil} : () & \mapsto & \text{nil} \\ \text{cons} : (x, z_1) & \mapsto & \text{insert}\{p\}(x, z_1) \end{array}\right|\!\!\right\}(z).$$

4.3.3 The map expression

Suppose we have a list of numbers to which we want to add some other number. We could certainly do this using the fold operation, however, there is a much more succinct expression for this using the map expression:

$$\text{list}(\mathcal{N}) \times \mathcal{N} \longrightarrow \text{list}(\mathcal{N}); (z, n) \mapsto \text{list}\{y \mapsto n + y\}(z).$$

Now this may seem to be a somewhat trivial thing to want to do but it is surprising how often one wishes to do something uniformly to every item of a list (or data structure).

For example suppose that one has two lists and one wants to form the list of all possible pairs in which the first record is an item from the first list and the second record is an item from the second list. This may be done as follows. First, form the list with each item a pair whose first record is an item of the first list and second record a copy of the second list, next for each of these pairs form the list of pairs whose records are respectively the item from the first list and an item from the second list. This will give a list of lists which must be flattened to obtain the list of pairs. The expression for this is:

$$\text{pairs} : \text{list}(X) \times \text{list}(Y) \longrightarrow \text{list}(X \times Y); (z_0, z_1) \mapsto \text{flatten}(\text{list}\{x_0 \mapsto \text{list}\{x_1 \mapsto (x_0, x_1)\}(z_1)\}(z_0)).$$

Now suppose $p : X \times Y \longrightarrow \text{Bool}$ is some proposition then we might have asked for a list of pairs for which $p(x, y) = \text{true}$. One way to obtain this list is as follows: first form all pairs then for each pair in the list apply the map which gives the empty list if the pair is outside p and the singleton list if it is inside p. Applying this uniformly to the list and then flattening will yield the desired result. Set

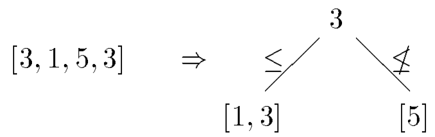
$$\text{filter}\{p\} : \text{list}(X) \longrightarrow \text{list}(X); z \mapsto \text{flatten}(\text{list}\{x \mapsto \left\{ \begin{array}{lcl} \text{true} & \mapsto & \text{cons}(x, \text{nil}) \\ \text{false} & \mapsto & \text{nil} \end{array} \right\} (p(x))\}(z))$$

then the map we want is $\text{pairs}; \text{filter}\{p\}$. These, of course, are the maps underlying list comprehension[Wad90]. This leads us to believe that it is suitable to add monads to charity because of the categorical basis of the system.

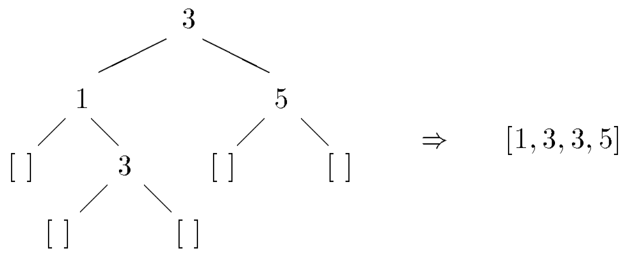
A further little example of the use of the map expression is given by the following: suppose we want to know how many times an item occurs in a list. We could form the list of pairs whose first record is the item and whose second record is an element of the list (map expression), then filter with respect to the equality test and determine the length of the resulting list.

4.4 Trees and Quicksort

Given a list of data and a predicate we have seen how we can order the data using an insertion sort. Here we show how charity can be used to sort a list of data using a method similar to Quicksort. The algorithm takes a list of unsorted data; removes the first element as a pivot piv; and divides the rest of the list l into two lists l_1 and l_2. Using a predicate p, each element x of l is put in l_1 if $p(x, piv)$ and l_2 otherwise. Using natural numbers and the relation $\leq$, the effect would be:



The process is then applied to the sublists. When all that remains are the pivots and empty sublists, the structure is a sorted binary tree with elements as the nodes and empty lists as the leaves. The nodes can then be gathered to produce the sorted list.



To implement this in charity we need to declare a datatype to hold the intermediate trees from above. This datatype is a binary tree where, if the input list is type $\text{list}(A)$ then the nodes contain elements of type A, and the leaves contain elements of type $\text{list}(A)$.

$$\begin{array}{rl} \textbf{data} & \text{btree}'(A) \longrightarrow X = \\ & \text{leaf} : \text{list}(A) \longrightarrow X \\ | & \text{node} : (X \times (A \times X)) \longrightarrow X \end{array}$$

This datatype definition suffices to implement the algorithm, but in order to present a map expression which has two phrases, the more general definition

$$\begin{array}{rl} \textbf{data} & \text{btree}(A, B) \longrightarrow X = \\ & \text{leaf} : A \longrightarrow X \\ | & \text{node} : (X \times (B \times X)) \longrightarrow X \end{array}$$

will be used. The difference between the definitions is that in btree the leaf may contain data not necessarily related to the type of data in the nodes. In fact, btree' is an instantiation of btree (ie. $\text{btree}'(A) \equiv \text{btree}(\text{list}(A), A)$).

Solving the problem from the bottom up, a map is needed which splits a list using the pivot and a predicate $p : A \times A \longrightarrow \text{Bool}$ (as in the insertion sort of section 4.3.2):

$$\text{split\_list}\{p\} : \text{list}(B) \times A \longrightarrow \text{list}(B) \times (A \times \text{list}(B));$$

$$(l, piv) \mapsto \left\{\!\!\left|\begin{array}{rcl} \text{nil} : () & \mapsto & (\text{nil}, (piv, \text{nil})) \\ \text{cons} : (a, (l_1, (piv, l_2))) & \mapsto & \left\{\begin{array}{rcl} \text{true} & \mapsto & (l_1, (piv, \text{cons}(a, l_2))) \\ \text{false} & \mapsto & (\text{cons}(a, l_1), (piv, l_2)) \end{array}\right\} (p(piv, a)) \end{array}\right|\!\!\right\} (l)$$

The state on which the fold operation acts is a triple: the first and third component are the lists containing elements “less than or equal to” and “greater than” the pivot piv respectively; the pivot is in the second component.

Next we define a map which extracts the pivot from a list and then divides the list using split_list.

$$\text{divide}\{p\} : \text{list}(A) \longrightarrow \text{btree}(\text{list}(A), A);$$

$$l \mapsto \left\{\begin{array}{rcl} \text{nil} & \mapsto & \text{leaf}(\text{nil}) \\ \text{cons}(a, \text{tail}) & \mapsto & \left\{ (l_1, (a, l_2)) \mapsto \text{node}(\text{leaf}(l_1), (a, \text{leaf}(l_2))) \right\} (\text{split\_list}\{p\}(tail, a)) \end{array}\right\} (l)$$

If the list l is empty an empty leaf is produced, otherwise a node with the pivot a and two leaves containing the sublists l_1 and l_2 is produced.

The map divide produces a tree type from the list type. Now we must apply divide to all the leaves of the tree for which we may use the map expression of btree:

$$\text{node\_it}\{p\} : \text{btree}(\text{list}(A), B) \longrightarrow \text{btree}(\text{btree}(\text{list}(A), A), B);$$

$$t \mapsto \text{btree}\left\{\begin{array}{rcl} x & \mapsto & \text{divide}\{p\}(x) \\ y & \mapsto & y \end{array}\right\} (t)$$

Notice how this map expression can perform two actions. This map will be applied to data of type $\text{btree}(\text{a}, \text{b})$ where a and b are instantiated instances of A and B. The first and second phrases describe the actions to be performed on the elements of type a and b respectively, thus divide will be applied to the leaves, and nodes will be left unchanged.

Since divide is applied to all the leaves of the tree, they will no longer contain lists of data, but will instead contain trees of data; this is also reflected in the type definition of node_it where list(A) becomes btree(list(A), A). To return the tree back to the proper form where leaves contain lists of data, a map btree_flatten is used:

$$\text{btree\_flatten} : \text{btree}(\text{btree}(A, B), B) \longrightarrow \text{btree}(A, B);$$

$$t \mapsto \left\{\!\!\left|\begin{array}{rcl} \text{leaf} : (b) & \mapsto & b \\ \text{node} : (c, (a, c')) & \mapsto & \text{node}(c, (a, c')) \end{array}\right|\!\!\right\}(t).$$

All of the leaves in the tree are replaced by the subtree that the leaf contains, effectively attaching the top node of the subtree back into the tree.

The maps node_it and btree_flatten can be used to provide one pass of the sort, thus these maps must be applied to the tree until all of the leaves contain empty lists (ie. the input elements are in sorted order in the tree). When the number of passes is at least the number of input elements this condition is met.

$$\text{apply}\{p\} : \text{list}(A) \longrightarrow \text{btree}(\text{list}(A), A);$$

$$l \mapsto \left\{\!\!\left|\begin{array}{rcl} \text{zero} : () & \mapsto & \text{leaf}(l) \\ \text{succ} : (t) & \mapsto & \text{btree\_flatten}(\text{node\_it}\{p\}(t)) \end{array}\right|\!\!\right\}(\text{length}(l))$$

The map length : list(A) $\longrightarrow \mathcal{N}$ returns the length of the list.

After there have been enough passes, the ordered tree can be gathered to produce an ordered list:

$$\text{gather} : \text{btree}(\text{list}(A), A) \longrightarrow \text{list}(A); t \mapsto \left\{\!\!\left|\begin{array}{rcl} \text{leaf} : (b) & \mapsto & b \\ \text{node} : (l_1, (a, l_2)) & \mapsto & \text{append}(l_1, cons(a, l_2)) \end{array}\right|\!\!\right\}(t).$$

The top level map, sort, requires a predicate p and the unordered list:

$$\text{sort}\{p\} : \text{list}(A) \longrightarrow \text{list}(A); l \mapsto \text{gather}(\text{apply}\{p\}(l)).$$

While the above algorithm implements the idea of Quicksort it fails to achieve the expected $n \log n$ complexity. The main reason for this is because node_it and btree_flatten always work over the whole tree when we only really need to be working on the leaves. The inefficiency is resolved by using a right btree in section 4.6.

4.5 Infinite lists

Up to this point all of the examples have used left datatypes exclusively. The right datatypes allow us to use possibly infinite data. Possibly the first infinite datatype that comes to mind is the infinite list.

$$\begin{array}{rl} \textbf{data} & S \longrightarrow \text{inflist}(A) = \\ & \text{head} : S \longrightarrow A \\ | & \text{tail} : S \longrightarrow S \end{array}$$

An infinite list, which takes a state of type S and produces an infinite list of type A, is described by the operations given for the head and tail.

4.5.1 The unfold expression

The unfold expression can be used to define the increasing infinite list of natural numbers (0, 1, 2, ...):

$$\text{nats} : 1 \longrightarrow \text{inflist}(\mathcal{N}); () \mapsto \left(\!\left| S \mapsto \begin{array}{l} \text{head} : S \\ \text{tail} : \text{succ}(S) \end{array} \right|\!\right) (\text{zero})$$

which says that the initial state is zero, the operation associated with head simply returns the state, and the operation associated with tail increments the state.

Both the head and the tail threads have operations which can work on the state S. The operation for head is not recursive (which is determined by the datatype declaration) and so returns a "value" by applying the map of the head thread to the state, while the operation for tail, which is recursive, produces a new state for the unfold expression.

$$\text{head}(\text{nats}) = \text{zero}$$

$$\text{head}(\text{tail}(\text{nats})) = \text{head}\left(() \mapsto \left(\!\left| S \mapsto \begin{array}{l} \text{head} : S \\ \text{tail} : \text{succ}(S) \end{array} \right|\!\right) (\underline{\text{succ}(\text{zero})}) \right) = \text{succ}(\text{zero})$$

We could also use the unfold expression to define an infinite list of the powers of 2 $(2^0, 2^1, 2^2 ...)$ by doubling the value of the previous state:

$$\text{powersof2} : 1 \longrightarrow \text{inflist}(\mathcal{N}); () \mapsto \left(\!\left| S \mapsto \begin{array}{l} \text{head} : S \\ \text{tail} : S + S \end{array} \right|\!\right) (\text{zero}).$$

4.5.2 The record expression

The record expression is useful for adding some constant data onto the front of an infinite list. If we wanted to add an extra zero to the front of nats we could do it simply by:

$$\text{zeronats} : 1 \longrightarrow \text{inflist}(\mathcal{N}); () \mapsto (\text{head} : \text{zero}, \text{tail} : \text{nats})$$

which would represent the infinite list (0, 0, 1, 2, ...).

4.5.3 The map expression

The effect of using the map expression is conceptually the same as the map expression on finite lists, in that it performs some operation on every element of the list. The difference is that the map expression on infinite lists must be lazy as it would take a very long time to do an operation on every element of an infinite list.

To get the infinite list of even numbers:

$$\text{evens} : 1 \longrightarrow \text{inflist}(\mathcal{N}); () \mapsto \text{inflist}\,\{x \mapsto x + x\}\,(\text{nats})$$

An alternative definition of getting the powersof2:

$$\text{powersof2} : 1 \longrightarrow \text{inflist}(\mathcal{N}); () \mapsto \text{inflist}\,\{x \mapsto 2^x\}\,(nats)$$

4.5.4 The Fibonacci Sequence

When working with finite objects an element of the Fibonacci sequence can be calculated by the following defining equations

$$\begin{aligned} \mathrm{fib}(0) &= 1 \\ \mathrm{fib}(1) &= 1 \\ \mathrm{fib}(n) &= \mathrm{fib}(n-1) + \mathrm{fib}(n-2), n \geq 2. \end{aligned}$$

Using an infinite list, we can conceptually create the whole sequence and then select the element that we want. Then the Fibonacci sequence is $(a_0, a_1, a_2, ...)$ such that:

$$\begin{aligned} a_0 &= 1 \\ a_1 &= 1 \\ a_n &= a_{n-1} + a_{n-2}, n \geq 2. \end{aligned}$$

Using an unfold, the above definition is given simply by

$$\mathrm{fib} : 1 \longrightarrow \mathrm{inflist}(\mathcal{N}); () \mapsto \left(\!\left| (x,y) \mapsto \begin{array}{rl} \mathrm{head}: & x \\ \mathrm{tail}: & (y, x+y) \end{array} \right|\!\right) (1,1)$$

which uses a pair for the state: the first and second components contain the value of the current element, and the next element respectively. The calculation of the new state (through the tail thread) is done by making the old next element the new current element, and calculating the new next element by adding the two old values together.

To get the nth value in the sequence (or any infinite list for that matter), we would do n tails and then a head:

$$\mathrm{get} : \mathcal{N} \times \mathrm{inflist}(A) \longrightarrow A; (n, L) \mapsto \mathrm{head}\left(\left\{\!\left| \begin{array}{rcl} \mathrm{zero}: () & \mapsto & L \\ \mathrm{succ}: (L) & \mapsto & \mathrm{tail}(L) \end{array} \right|\!\right\} (n) \right).$$

4.5.5 The Ackermann's function

Using an infinite list of infinite lists to build an infinite table, we can define the Ackermann's function. This is significant because the initial datatypes had restricted us to the domain of primitive recursive functions, but with the power of final datatypes we can define a combinator which is not primitive recursive.

The defining equations for the Ackermann's function are:

$$\begin{aligned} \mathrm{ack}(0, n) &= s(n) && (1) \\ \mathrm{ack}(s(m), 0) &= \mathrm{ack}(m, 1) && (2) \\ \mathrm{ack}(s(m), s(n)) &= \mathrm{ack}(m, \mathrm{ack}(s(m), n)) && (3) \end{aligned}$$

Letting m and n be the column and row index respectively, equation 1 tells us that column 0 is an infinite list where the ith value is $i + 1$:

	0	$\cdots$
0	1	$\cdots$
1	2	$\cdots$
2	3	$\cdots$
$\vdots$	$\vdots$	$\ddots$

That is column 0 is the infinite list of successive numbers starting from 1:

$$\text{col}_0 = \text{tail}(\text{nats})$$

Equation 2 tells us that the values in row 0 of a column are the same as the value in row 1 of the previous column.

	0	1	$\cdots$
0	1	**2**	$\cdots$
1	**2**	?	$\cdots$
2	3	?	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

Finally, equation 3 tells us that the value in position $(s(m), s(n))$ is the same as the value in position (m, i) where i is the value in position $(s(m), n)$:

	0	1	$\cdots$
0	1	$\underline{2}$	$\cdots$
1	2	**3**	$\cdots$
2	**3**	?	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

Thus, the value in position (1,1) is determined by taking the value 2 from position (1,0) and using it as an index into the previous column.

Since, all values are determined by previously calculated values, we can generate the Ackermann's table:

	0	1	2	3	$\cdots$
0	1	2	3	5	$\cdots$
1	2	3	5	13	$\cdots$
2	3	4	7	29	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

To calculate column m we use the values from column $m-1$:

$$\text{col}_m : \text{inflist}(\mathcal{N}) \longrightarrow \text{inflist}(\mathcal{N});$$

$$L \mapsto \left(\left| (nxt, L) \mapsto \begin{array}{rl} \text{head}: & \text{head}(L) \\ \text{tail}: & \{newL \mapsto (\text{head}(newL) \dot{-} nxt, newL)\} \\ & \left(\left\{ \begin{array}{lcl} \text{zero}:() & \mapsto & L \\ \text{succ}: s & \mapsto & \text{tail}(s) \end{array} \right\} (nxt) \right) \end{array} \right| \right) (\text{pred}(\text{head}(\text{tail}(L))), \text{tail}(L))$$

The variable L is the previous column in the table. The variable nxt is used to store the offset (number of tails needed) to the next value.

The Ackermann's table is then simply the infinite list of all the columns:

$$\text{ack\_table} : 1 \longrightarrow \text{inflist}(\text{inflist}(\mathcal{N}));$$
$$() \mapsto \left(\!\left| L \mapsto \quad \text{head} : L \, \text{tail} : \text{col}_m(L) \right|\!\right) (\text{col}_0)$$

To get the value in position (m,n), we could use the get combinator to first find the proper column, and then the proper row in that column:

$$\text{ack} : \mathcal{N} \times \mathcal{N} \longrightarrow \mathcal{N}; (m,n) \mapsto \text{get}(n, \text{get}(m, \text{ack\_table}))$$

4.6 Quicksort revisited

Here we show how the Quicksort can be more efficiently implemented using a final datatype to hold the partially sorted data.

Again, the datatype to store the data is a binary tree, the difference this time is that the binary tree is defined as a final datatype and may be infinite (although our trees will be finite).

$$\begin{array}{ll} \textbf{data} & S \longrightarrow \text{rbtree}(A,B) = \\ & \text{look} : S \longrightarrow S \times (B \times S) + A \end{array}$$

Every time the destructor look is applied to the tree, a node (b_0) or a leaf (b_1) value will be returned.

Now by using the pivot map defined above, a map, tree, which creates a sorted binary tree given a predicate and input list can be defined:

$$\text{tree}\{p\} : \text{list}(A) \longrightarrow \text{rbtree}(1,A); (l) \mapsto (\!| x \mapsto \text{look} : \text{pivot}\{p\}(x) |\!) (l)$$

So the creation of the sorted tree is quick, the real problem comes when we try to convert the tree to a sorted list. This was easy when the tree was an initial datatype because we could use the fold operation to iterate over it. The procedure that we will use to collect the tree will be to create a state consisting of three components: the accumulated list (L), the current part of the tree we are looking at (T), and the dump (D). We will then manipulate the state as follows:

- If we are looking at the node of a tree on T then we will push the node value and the right subtree onto the dump, and continue with the left subtree on T:

$$(L, (\mathit{Tleft}, \mathit{value}, \mathit{Tright}), D) \Rightarrow (L, \mathit{Tleft}, (\mathit{value}, \mathit{Tright}).D)$$

- If we are looking at a leaf on T then we will pop an element (a,T) from the dump; the value a will be added to L and we will work on T:

$$(L, (), (a,T).D) \Rightarrow (a.L, T, D)$$

To calculate the number of times that the above operations must be done, we note that for a list of length n there will be n nodes and $n+1$ leaves, so we will require $2n+1$ pushs and pops to collect up the tree.

The collect algorithm follows:

$$\text{pop} : \text{list}(A) \times \text{list}(A \times \text{rbtree}(1, B)) \longrightarrow \text{list}(A) \times (\text{rbtree}(1, B) \times \text{list}(A \times \text{rbtree}(1, B)));$$

$$(acc, dump) \mapsto \left\{ \begin{array}{rcl} \text{nil}() & \mapsto & (acc, ((\text{look} : \text{b}_1()), [])) \\ \text{cons}((a,t), dump') & \mapsto & (\text{cons}(a, acc), (t, dump')) \end{array} \right\} (dump)$$

The push is so straightforward that we directly put it into the collect map.

$$\text{collect} : \text{rbtree}(1, A) \times \mathcal{N} \longrightarrow \text{list}(A) \times (\text{rbtree}(1, A) \times \text{list}(A * \text{rbtree}(1, A)));$$

$$(tree, size) \mapsto \left\{\left| \begin{array}{l} \text{zero} : () \quad \mapsto \quad ([], (tree, [])) \\ \text{succ} : (L, (T, D)) \mapsto \\ \qquad \left\{ \begin{array}{rcl} \text{b}_1() & \mapsto & \text{pop}(L, D) \\ \text{b}_0(Tl, (a, Tr)) & \mapsto & (L, (Tl, \text{cons}((a, Tr), D))) \end{array} \right\} (\text{look}(T)) \end{array} \right|\right\} (size)$$

Since the collect map makes only one pass over the whole tree, as opposed to the n passes that the previous Quicksort made, we can see that this sort is more efficient.

To calculate the expected complexity of the above algorithm, we assume that any permutation of the input is equally likely. The collect map takes time kn for some constant k. The generation of the tree depends on the unsorted input list: if we select the ith smallest element as the pivot, it will take time $T(i-1)$ and $T(n-i)$ to sort the subtrees. This gives us the relationship:

$$T(n) \leq kn + \frac{1}{n} \sum_{i=1}^{n} (T(i-1) + T(n-i)).$$

In [AVA74] this is shown to have $\mathbf{O}(n \log n)$ as expected.

5 Proving equality of maps

The term logic has rules of inference for handling equations which make it logically equivalent to the category itself. The purpose of this section is to show informally how the term logic can be used to prove equivalence of programs.

5.1 Case analysis

A common method of proving that two maps are equal in a distributive category is by a **case analysis**. This involves splitting the domain into coproduct components over each of which the argument for equality can be made. While this works in any category with coproducts, it works particularly well when the coproducts are universal because the components can be chosen to reflect the splitting on the intermediate domains which occur naturally in the proof.

A case analysis can be lain out somewhat like a decision tree, allowing each subcase to be further split in a non–uniform way in the manner required to complete the proof down that branch. Often the branchings can be arranged naturally using propositions (maps $p : X \longrightarrow 1+1$) and the splitting induced by assuming in turn that they are true or false ($p(x) = \text{true}$ and $p(x) = \text{false}$).

As a trivial example consider the following two formulations of $\wedge$:

$$\wedge : \mathrm{Bool} \times \mathrm{Bool} \longrightarrow \mathrm{Bool}; (x_0, x_1) \mapsto \left\{ \begin{array}{lcl} \mathrm{true} & \mapsto & x_1 \\ \mathrm{false} & \mapsto & \mathrm{false} \end{array} \right\} (x_0)$$

$$\wedge' : \mathrm{Bool} \times \mathrm{Bool} \longrightarrow \mathrm{Bool}; (x_0, x_1) \mapsto \left\{ \begin{array}{lcl} \mathrm{true} & \mapsto & x_0 \\ \mathrm{false} & \mapsto & \mathrm{false} \end{array} \right\} (x_1)$$

To prove they are the same we perform the following case analysis:

Case 1: $x_0 = \mathrm{true}$.

This means that $x_0 \wedge x_1 = x_1$ and it remains to do a case analysis on x_1 to obtain the values of $\wedge'$.

Case 1.1: $x_1 = \mathrm{true}$.

$$x_0 \wedge' x_1 = x_0 = \mathrm{true} = x_1 = x_0 \wedge x_1.$$

Case 1.2: $x_1 = \mathrm{false}$.

$$x_0 \wedge' x_1 = \mathrm{false} = x_1 = x_0 \wedge x_1.$$

Case 2: $x_0 = \mathrm{false}$.

This means that $x_0 \wedge x_1 = \mathrm{false}$ and it remains to do a case analysis on x_0 to obtain the values for $\wedge'$.

Case 2.1: $x_1 = \mathrm{true}$.

$$x_0 \wedge' x_1 = x_0 = \mathrm{false} = x_0 \wedge x_1.$$

Case 2.2: $x_1 = \mathrm{false}$.

$$x_0 \wedge' x_1 = \mathrm{false} = x_0 \wedge x_1.$$

Below we will consider a less trivial example.

5.2 Structural induction

Particularly important is the case analysis associated with an initial datatype. Given the datatype declaration of section 3.2, suppose we wish to prove that

$$f, g : L(A) \times X \longrightarrow Z$$

are equal. It suffices to prove that their equalizer is the whole of $L(A) \times X$. Let (E, e) be the equalizer of f and g, then it suffices to prove that e is an isomorphism, which given that e is necessarily monic is established if we can find an a with $a; e = 1$. This in turn is provided if we can show that E is "closed under the action" in the sense

$$\begin{tikzcd}
F_i(A,E)\times X \arrow[r, dotted, "e_i"] \arrow[d, "F_i(1,e)\times 1"'] & E \arrow[dd, "e"] \\
F_i(A,L(A)\times X)\times X \arrow[d, "F_i(1,\mathrm{p}_0)\times 1"'] & \\
F_i(A,L(A))\times X \arrow[r, "c_i\times 1"'] & L(A)\times X
\end{tikzcd}$$

that there is an $e_i : F_i(A,E)\times X \longrightarrow E$ such that $e_i; e = (F(1,e;\mathrm{p}_0);c_i)\times 1$. If we have such a factorization then we have

$$\begin{tikzcd}
F_i(A,L(A))\times X \arrow[r, "c_i"] \arrow[d, "\langle\theta_R^{F_i},\mathrm{p}_1\rangle"'] & L(A)\times X \arrow[dd, "a"] \\
F_i(A,L(A)\times X)\times X \arrow[d, "F_i(1,a)\times 1"'] & \\
F_i(A,E)\times X \arrow[r, "e_i"] \arrow[d, "F_i(1,e;\mathrm{p}_0)\times 1"'] & E \arrow[d, "e;\mathrm{p}_0"] \\
F_i(A,L(A))\times X \arrow[r, "c_i\times 1;\mathrm{p}_0"'] & L(A)
\end{tikzcd}$$

in which $a;e;\mathrm{p}_0 = \mathrm{p}_0$ by the uniqueness of the fold map. But also $a;e;\mathrm{p}_1$ is determined by

$$\begin{aligned}
(c_i\times 1);a;e;\mathrm{p}_1 &= \langle\theta_R^{F_i},\mathrm{p}_1\rangle;((F_i(1,a;e;\mathrm{p}_0);c_i)\times 1);\mathrm{p}_1 \\
&= \langle\theta_R^{F_i},\mathrm{p}_1\rangle;\mathrm{p}_1 \\
&= \mathrm{p}_1
\end{aligned}$$

so certainly $a;e;\mathrm{p}_1 = \mathrm{p}_1$. But this makes $a;e=1$ and so e is an isomorphism.

Specializing this argument to that of determining whether $f,g:\mathcal{N}\times X \longrightarrow Z$ are equal shows that it suffices to prove that $f(0,x) = g(0,x)$ and that if $f(n,x) = g(n,x)$ then $f(s(n),x) = g(s(n),x)$. Similarly to determine whether $f,g:\mathrm{list}(A)\times X \longrightarrow Z$ are equal it suffices to prove $f(\mathrm{nil},x) = g(\mathrm{nil},x)$ and if $f(z,x) = g(z,x)$ then $f(\mathrm{cons}(a,z),x) = g(\mathrm{cons}(a,z),x)$.

As an example we shall now sketch a proof that the naive reverse is equal as a map to the "fast" reverse. To do this we start by proving a lemma:

Let

$$q(y_0,y_1) = \left\{\begin{array}{rcl} \mathrm{nil} & \mapsto & (y_0,y_1) \\ \mathrm{cons}(a,y_1') & \mapsto & (\mathrm{cons}(a,y_0),y_1') \end{array}\right\}(y_1)$$

then clearly $\text{reverse}'(z) = \text{p}_0(q^{\text{length}(z)}(\text{nil}, z))$.

Lemma 5.1

$$q^{\text{length}(z_0)}(z_1, z_2) = (\text{append}(\text{p}_0(q^{\text{length}(z_0)}(\text{nil}, z_2)), z_1), \text{tail}^{\text{length}(z_0)}(z_2)).$$

Proof. We shall do a structural induction on z_0:

Case 1: $z_0 = \text{nil}$.

The equality is immediate.

Case 2: $z_0 = \text{cons}(a, z_0')$ and the result holds for z_0'.

$$\begin{aligned} q^{\text{length}(\text{cons}(a,z_0'))}(z_1, z_2) &= q(q^{\text{length}(z_0')}(z_1, z_2)) \\ &= q(\text{append}(\text{p}_0(q^{\text{length}(z_0')}(\text{nil}, z_2)), z_1), \text{tail}^{\text{length}(z_0')}(z_2)) \end{aligned}$$

To complete the proof we must pull the append through an application of q. To show that this may be done we resort to a case analysis on the form of $\text{tail}^{\text{length}(z_0')}(z_2)$.

Case 2.1: $\text{tail}^{\text{length}(z_0')}(z_2) = \text{nil}$.

then the effect of q is the identity and the result holds.

Case 2.2: $\text{tail}^{\text{length}(z_0')}(z_2) = \text{cons}(a', z_2')$.

Setting $w = \text{p}_0(q^{\text{length}(z_0')}(\text{nil}, z_2))$ we have:

$$\begin{aligned} q(\text{append}(w, z_1), \text{cons}(a', z_2')) &= (\text{cons}(a', \text{append}(w, z_1)), z_2') \\ &= (\text{append}(\text{cons}(a', w), z_1), z_2') \end{aligned}$$

Finally $\text{cons}(a', w) = \text{p}_0(q^{\text{length}(z_0)}(\text{nil}, z_2))$.

□

So now we have:

Proposition 5.2

$$\text{reverse}(z) = \text{reverse}'(z).$$

Proof. By structural induction on z:

Case 1: $z = \text{nil}$.

Clear.

Case 2: $z = \text{cons}(a, z')$ and the result holds for z'.

We have using the above lemma:

$$\begin{aligned} \text{reverse}(\text{cons}(a, z')) &= \text{append}(\text{reverse}(z'), \text{cons}(a, \text{nil})) \\ &= \text{append}(\text{reverse}'(z'), \text{cons}(a, \text{nil})) \\ &= \text{append}(\text{p}_0(q^{\text{length}(z')}(\text{nil}, z'), \text{cons}(a, \text{nil})) \\ &= \text{p}_0(q^{\text{length}(z')}(\text{cons}(a, \text{nil}), z')) \\ &= \text{p}_0(q^{\text{length}(\text{cons}(a,z'))}(\text{nil}, \text{cons}(a, z'))) \\ &= \text{reverse}'(\text{cons}(a, z')). \end{aligned}$$

$\square$

Arguments by structural induction are already very familiar: it is significant that they can be used in this setting. However, the real significance of this setting is that there are many other proof techniques available of the "theorems for free" kind[Wad89].

6 Evaluating Categorical Combinators

It was Tatsuya Hagino's contribution to show that categories constructed from datatypes can be used directly as programming languages. The languages that result have some rather special properties: the principle one being that one cannot write non-terminating programs in them.

6.1 Eager evaluation in a category

In order to see how evaluation comes about in a category it is useful to trace the ideas through some simpler steps in order to arrive at the by-value machine currently used in charity.

Towards this end we explore a simple categorical rewriting system. It relies on the fact that any composition of primitive symbols of the category can always be associatively rearranged into a left–associated expression followed by a right–associated expression. The place where these expressions meet is denoted by a $\star$ and will indicate the position of the *rewriting head*

$$(..(((1; a_1); a_2); a_3); \ldots; a_i) \star (a_{i+1}; (\ldots; (a_{n-2}; (a_{n-1}; (a_n; 1)))..).$$

Any computer scientist will instantly realize that the way to implement this is as two stacks. The rewriting head then starts on the left and traverses the composition once, to the right, by popping things off the right stack and pushing them onto the left stack. Whenever a situation is detected to be one in which a reduction can be performed the reduction is performed rather than the popping and pushing. This detection process will involve looking *only* at the top cells of each stack.

Consider the following rules:

$$\begin{array}{ll} a_1; f \Longrightarrow a_2 & a_1; g \Longrightarrow a_3 \\ a_2; f \Longrightarrow a_3 & a_2; g \Longrightarrow a_1 \\ a_3; f \Longrightarrow a_1 & a_3; g \Longrightarrow a_1 \end{array}$$

The following is a chain rewriting on the term:

$$a_1; f; g; g$$

as follows:

$$\begin{array}{ll} \star a_1; f; g; g & \\ \quad \Longrightarrow & a_1 \star f; g; g \\ \quad \Longrightarrow & a_2 \star g; g \\ \quad \Longrightarrow & a_1 \star g \\ \quad \Longrightarrow & a_3 \star \end{array}$$

In fact, this is a good stab at what a full blown **abstract machine** for rewriting any functional language looks like. Traditionally the left stack is called the **value** stack and the right stack is called the **code** stack. Obviously this is still overly simple as sometimes we will meet a situation in which a major sub-reduction must be performed before one can continue with the rewriting of the main chain. To facilitate this, one just needs one more stack, traditionally called the **dump**, on which to push intermediate states of the calculation to which one must return. With this addition, a full blown categorical abstract machine is obtained.

6.2 Using a predistributive category

To illustrate how the abstract machine works consider the rewriting rules associated with a predistributive category.

$$\begin{array}{rcl} z; ! & \Longrightarrow & ! \\ \langle x, y\rangle; \mathrm{p}_0 & \Longrightarrow & x \\ \langle x, y\rangle; \mathrm{p}_1 & \Longrightarrow & y \\ \langle z; \mathrm{b}_0, x\rangle; \langle f \mid g\rangle & \Longrightarrow & \langle z, x\rangle; f \\ \langle z; \mathrm{b}_1, x\rangle; \langle f \mid g\rangle & \Longrightarrow & \langle z, x\rangle; g \\ z; \langle x, y\rangle & \Longrightarrow & \langle z; x, z; y\rangle \end{array}$$

The transitions for the corresponding **abstract machine** are:

	value	**code**	**dump**		**value**	**code**	**dump**
1	v	$!.c$	d	$\longrightarrow$	$!$	c	d
2	$\langle v_0, v_1\rangle$	$\mathrm{p}_0\,.c$	d	$\longrightarrow$	v_0	c	d
3	$\langle v_0, v_1\rangle$	$\mathrm{p}_1\,.c$	d	$\longrightarrow$	v_1	c	d
4	$\langle v_0 . \mathrm{b}_0, v_1\rangle$	$\langle c_0 \mid c_1\rangle.c$	d	$\longrightarrow$	$\langle v_0, v_1\rangle$	c_0	$\mathrm{c}(c).d$
5	$\langle v_0 . \mathrm{b}_1, v_1\rangle$	$\langle c_0 \mid c_1\rangle.c$	d	$\longrightarrow$	$\langle v_0, v_1\rangle$	c_1	$\mathrm{c}(c).d$
6	v	$\langle c_0, c_1\rangle.c$	d	$\longrightarrow$	v	c_0	$\mathrm{pr}_0(v, c_1, c).d$
7	v_0	$[\,]$	$\mathrm{pr}_0(v, c_1, c).d$	$\longrightarrow$	v	c_1	$\mathrm{pr}_1(v_0, c).d$
8	v_1	$[\,]$	$\mathrm{pr}_1(v_0, c).d$	$\longrightarrow$	$\langle v_0, v_1\rangle$	c	d
9	v	$u.c$	d	$\longrightarrow$	$v.u$	c	d
10	v	$[\,]$	$\mathrm{c}(c)$	$\longrightarrow$	v	c	d
11	v	$[\,]$	$[\,]$	$\longrightarrow$		**STOP**	

Transitions 1 to 5 correspond closely to the rewriting rules.

Transitions 6 to 8 give us the distributive rule. Transition 6 says that when we encounter a pair, evaluate the first component of the pair and for the product, hold the second component on the dump for later evaluation. When the first component has been evaluated, we will reach transition 7, which stores away the result for the first component of the pair and evaluates the second component. Transition 8 puts the reduced pair on the value stack and resumes execution with the code following the pair.

Transitions 9 to 11 are generic. Transition 9 says that if there is no action associated with the value on the top of the code stack (ie. b_0 and b_1), push it onto the value stack. Transition 10 says that if the code stack is empty and there is a continuation on the dump then execute the continuation. Transition 11 indicates that the rewriting is complete: the result is on the value stack.

6.3 Adding the initial datatypes

When we add initial datatypes there are some associated rewrite rules that must be added to the system. With each datatype (defined as in section 3.2) come constructors, a case, a fold and a map.

The new constructors are handled by transition 9.

The case combinator gives the simplest rewrite rule. Given a list of maps, we simply chose the map that positionally corresponds to the constructor on the value stack.

$$\langle v_0; c_i, v_1\rangle; \mathrm{case}^L\{f_1, ..., f_n\} \Longrightarrow \langle v_0, v_1\rangle; f_i$$

The fold is more complex. From the diagram for the fold we obtain a rewrite rule going from the expression $c_i \times 1; \mathrm{fold}^L$ to the more defined expression $\langle \theta_R^{E_i}, \mathrm{p}_1\rangle; E_i(1, \mathrm{fold}^L) \times 1; g_i$. The latter map can be written using the map combinator as $\langle \mathrm{map}^{E_i}\{\mathrm{p}_0, \mathrm{fold}^L\{g_1, ..., g_n\}\}, \mathrm{p}_1\rangle; g_i$. This gives us:

$$\langle v_0; c_i, v_1\rangle; \mathrm{fold}^L\{g_1, ..., g_n\} \Longrightarrow \langle v_0, v_1\rangle; \langle \mathrm{map}^{E_i}\{\mathrm{p}_0, \mathrm{fold}^L\{g_1, ..., g_n\}\}, \mathrm{p}_1\rangle; g_i$$

From the defining diagram for map we get:

$$\langle v_0; c_i, v_1\rangle; \mathrm{map}^L\{h_1, ..., h_m\} \Longrightarrow \langle v_0, v_1\rangle; \mathrm{map}^{E_i}\{h_1, ..., h_m, \mathrm{map}^L\{h_1, ..., h_m\}\}; c_i.$$

The corresponding transitions for the above rules are

	value	**code**	**dump**		**value**	**code**	**dump**
12	$\langle v_0.c_i, v_1\rangle$	$\mathrm{case}^L\{f_1, ..., f_n\}.c$	d	$\longrightarrow$	$\langle v_0, v_1\rangle$	f_i	$\mathrm{c}(c).d$
13	$\langle v_0.c_i, v_1\rangle$	$\mathrm{fold}^L\{g_1, ..., g_n\}.c$	d	$\longrightarrow$	$\langle v_0, v_1\rangle$	$\langle \mathrm{map}^{E_i}\{\mathrm{p}_0, \mathrm{fold}^L\{g_1, ..., g_n\}\},$ $\mathrm{p}_1\rangle.g_i$	$\mathrm{c}(c).d$
14	$\langle v_0.c_i, v_1\rangle$	$\mathrm{map}^L\{h_1, ..., h_m\}.c$	d	$\longrightarrow$	$\langle v_0, v_1\rangle$	$\mathrm{map}^{E_i}\{h_1, ..., h_m,$ $\mathrm{map}^L\{h_1, ..., h_m\}\}.c_i$	$\mathrm{c}(c).d$

For the factorizers, a table is needed which holds the name of the constructor c_i, its type, and its corresponding position in the definition of L. The case combinator uses the position to select the proper f_i from the list. The fold uses the position to select the proper g_i and then uses the type information to do the proper substitutions (ie. to use p_0 or $\mathrm{fold}^L\{g_1, ..., g_n\}$). For the map combinator, the type information is used to select the proper h_j or $\mathrm{map}^L\{h_1, ..., h_m\}$.

6.4 Adding finite products

The rewrite rules for the finite products of which there are only three, are as follows:

$$\begin{aligned} \langle v_0, v_1\rangle; \mathrm{map}^1 &\Longrightarrow v_0 \\ \langle v_0, v_1\rangle; \mathrm{map}^{\mathrm{Id}}\{f\} &\Longrightarrow \langle v_0, v_1\rangle; f \\ \langle\langle v_0, v_1\rangle, v_2\rangle; \mathrm{map}^{\times}\{f, g\} &\Longrightarrow \langle\langle v_0, v_2\rangle, \langle v_1, v_2\rangle\rangle; f \times g \end{aligned}$$

These produce the following machine transitions:

	value	**code**	**dump**		**value**	**code**	**dump**
15	$\langle v_0, v_1\rangle$	$\mathrm{map}^1.c$	d	$\longrightarrow$	v_0	c	d
16	v	$\mathrm{map}^{\mathrm{Id}}\{f\}.c$	d	$\longrightarrow$	v	f	$\mathrm{c}(c).d$
17	$\langle\langle v_0, v_1\rangle, v_2\rangle$	$\mathrm{map}^{\times}\{f, g\}.c$	d	$\longrightarrow$	$\langle v_0, v_2\rangle$	f	$\mathrm{pr}_0(\langle v_1, v_2\rangle, g, c).d$

6.5 Adding the final datatypes

For the final datatypes, rules must be added for the destructors. The unfold, record and map are inactive combinators and so are handled by transition 9. The destructor behaves differently depending on what type of data it is destructing.

$$\begin{aligned}
\mathrm{unfold}^R\{f_1,...,f_n\}; d_i &\implies \langle f_i, \mathrm{p}_1\rangle; \mathrm{map}^{F_i}\{\mathrm{p}_0, \mathrm{unfold}^R\{f_1,...,f_n\}\} \\
\mathrm{record}^R\{g_1,...,g_n\}; d_i &\implies g_i \\
\mathrm{map}^R\{h_1,...,h_m\}; d_i &\implies d_i \times 1; \mathrm{map}^{F_i}\{h_1,...,h_m, \mathrm{map}^R\{h_1,...,h_m\}\}
\end{aligned}$$

The corresponding transitions for the rules are:

	value	**code**	**dump**		**value**	**code**	**dump**
18	$v.\,\mathrm{unfold}^R\{f_1,...,f_n\}$	$d_i.c$	d	$\longrightarrow$	v	$\langle f_i, \mathrm{p}_1\rangle.\,\mathrm{map}^{F_i}\{\mathrm{p}_0,$ $\mathrm{unfold}^R\{f_1,...,f_n\}\}$	$\mathrm{c}(c).d$
19	$v.\,\mathrm{record}^R\{g_1,...,g_n\}$	$d_i.c$	d	$\longrightarrow$	v	g_i	$\mathrm{c}(c).d$
20	$v.\,\mathrm{map}^R\{h_1,...,h_m\}$	$d_i.c$	d	$\longrightarrow$	v	$\langle p0.d_i, p1\rangle.\,\mathrm{map}^{F_i}\{h_1,...,h_m,$ $\mathrm{map}^R\{h_1,...,h_m\}\}$	$\mathrm{c}(c).d$

7 Translating the term logic

The categorical combinators (despite what some category theorist seem to think) are totally unreadable! We really want to use the term logic as the programming language as this is much more friendly. However, to do so we need a translation process from the term logic into the combinators for evaluation. This raises all sorts of nasty issues like how one does an efficient translation. However, one of the essential points to note is that unlike the initial stabs at translating into Curry's combinators (which turned out to be exponential and only after considerable work was brought down by Turner to polynomial[Tur79b, Tur79a] and later Statman to $n \log n$), we enter the competition with an $n \log n$ translation. As this is the theoretical limit (don't forget the constants where there is room for improvement) we can be reasonably happy.

We shall set up the term logic for a predistributive category with datatypes and then describe the translation to combinators. A proof of the translation is given in [CS92]. We assume that the datatype declarations are of the form given in section 3.2.

7.1 Variable bases

For each type in the category we shall suppose that we have a set of variables (in fact the type is inferred) $\{x, y, z, ...\}$.

A **variable base** is then defined by

- $()$ is a variable base of type 1,
- If x is a variable then x is a variable base with type type(x),
- If v_0 and v_1 are variable bases *with no variables in common* then (v_0, v_1) is a variable base where

$$\mathrm{type}((v_0, v_1)) = \mathrm{type}(v_0) \times \mathrm{type}(v_1).$$

7.2 Terms

A **term** is defined by:

- $()$ is a term of type $\mathrm{type}(()) = 1$,
- If t is a term where $\mathrm{type}(t) = X \times Y$ then $\mathrm{p}_0(t)$ and $\mathrm{p}_1(t)$ are terms where $\mathrm{type}(\mathrm{p}_0(t)) = X$ and $\mathrm{type}(\mathrm{p}_1(t)) = Y$,
- If t_0 and t_1 are terms then (t_0, t_1) is a term where $\mathrm{type}((t_0, t_1)) = \mathrm{type}(t_0) \times \mathrm{type}(t_1)$,
- If w is a variable base and t is a term where $\mathrm{type}(w) = \mathrm{type}(t)$ then

  $$\{w \mapsto t'\}(t)$$

  is a term of type $\mathrm{type}(t')$, and the variables in w are bound in t'.
- If t is a term $\mathrm{b}_0(t)$ and $\mathrm{b}_1(t)$ are terms where $\mathrm{type}(\mathrm{b}_0(t)) = \mathrm{type}(t) + X$ and $\mathrm{type}(\mathrm{b}_1(t)) = X + \mathrm{type}(t)$ (where X is an indeterminate type).
- If t is a term where $\mathrm{type}(t) = A_0 + A_1$ and v_0 and v_1 are variable bases where $\mathrm{type}(v_i) = A_i$, and t_0 and t_1 are terms where $\mathrm{type}(t_0) = \mathrm{type}(t_1) = B$ then

  $$\left\{ \begin{array}{ccc} \mathrm{b}_0(v_0) & \mapsto & t_0 \\ \mathrm{b}_1(v_1) & \mapsto & t_1 \end{array} \right\}(t)$$

  is a term of type B. The variables in v_0 and v_1 are bound in t_0 and t_1 respectively.
- If $t_1, ..., t_n$ are terms where $\mathrm{type}(t_i) = E_i(A, L(A))$ then $c_1(t_1), ..., c_n(t_n)$ are terms where $\mathrm{type}(c_i(t_i)) = L(A)$.
- If t is a term where $\mathrm{type}(t) = L(A)$ and $v_1, ..., v_n$ are variable bases where $\mathrm{type}(v_i) = E_i(A, L(A))$ and $t_1, ..., t_n$ are terms where $\mathrm{type}(t_1) = ... = \mathrm{type}(t_n) = B$ then

  $$\left\{ \begin{array}{ccc} \mathrm{c}_1(v_1) & \mapsto & t_1 \\ & \vdots & \\ \mathrm{c}_n(v_n) & \mapsto & t_n \end{array} \right\}(t)$$

  is a term (the case expression) of type B. The variables in $v_1, ..., v_n$ are bound in $t_1, ..., t_n$ respectively.
- If t is a term where $\mathrm{type}(t) = L(A)$ and $v_1, ..., v_n$ are variable bases where $\mathrm{type}(v_i) = E_i(A, X)$ and $t_1, ..., t_n$ are terms where $\mathrm{type}(t_1) = ... = \mathrm{type}(t_n) = X$ then

  $$\left\{\!\left| \begin{array}{ccc} \mathrm{c}_1 : v_1 & \mapsto & t_1 \\ & \vdots & \\ \mathrm{c}_n : v_n & \mapsto & t_n \end{array} \right|\!\right\}(t)$$

  is a term (the fold) of type X. The variables in $v_1, ..., v_n$ are bound in $t_1, ..., t_n$ respectively.

- If t is a term where $\text{type}(t) = L(A_1, ..., A_m)$ and $v_1, ..., v_m$ are variable bases where $\text{type}(v_j) = A_j$ and $t_1, ..., t_m$ are terms where $\text{type}(t_j) = B_j$ then

$$L \left\{ \begin{array}{ccc} v_1 & \mapsto & t_1 \\ & \vdots & \\ v_m & \mapsto & t_m \end{array} \right\} (t)$$

is a term (the map expression) of type $L(B_1, ..., B_m)$. The variables in $v_1, ..., v_m$ are bound in $t_1, ..., t_m$ respectively.

- If t is a term where $\text{type}(t) = S$ and v is a variable base where $\text{type}(v) = S$ and $t_1, ..., t_n$ are terms where $\text{type}(t_j) = F_j(A, S)$ then

$$\left(\!\!\left| v \mapsto \begin{array}{ccc} d_1 & : & t_1 \\ & \vdots & \\ d_n & : & t_n \end{array} \right|\!\!\right) (t)$$

is a term (the unfold) of type $R(A)$. The variables in v are bound in $t_1, ..., t_n$ respectively.

- If $t_1, ..., t_n$ are terms where $\text{type}(t_j) = F_j(A, R(A))$ then

$$\left(\begin{array}{ccc} d_1 & : & t_1 \\ & \vdots & \\ d_n & : & t_n \end{array} \right)$$

is a term (the record) of type $R(A)$.

7.3 Abstracted maps

A program is not a term but an **abstracted map** this is a pair

$$\{v \mapsto t\}$$

where v is a variable base containing all the free variables of the term t.

The abstracted maps are what we must translate into combinators so that we can evaluate them.

7.4 Translation to categorical combinators

The translation $\mathcal{T}$ from the term logic is described as follows:

- $\mathcal{T}[v \mapsto ()] = !$,
- $\mathcal{T}[x \mapsto x] = 1$,
- $\mathcal{T}[(v_0, v_1) \mapsto x] = \mathrm{p}_i; \mathcal{T}[v_i \mapsto x]$where $i = 0$ if x occurs in v_0 otherwise $i = 1$,
- $\mathcal{T}[v \mapsto \{w \mapsto t'\}(t)] = \langle \mathcal{T}[v \mapsto t], 1 \rangle; \mathcal{T}[(w, v) \mapsto t']$,

- $\mathcal{T}[v \mapsto \mathrm{p}_i(t)] = \mathcal{T}[v \mapsto t]; \mathrm{p}_i$ for $i = 0, 1$,
- $\mathcal{T}[v \mapsto (t_0, t_1)] = \langle \mathcal{T}[v \mapsto t_0], \mathcal{T}[v \mapsto t_1] \rangle$,
- $\mathcal{T}[v \mapsto \mathrm{b}_i(t)] = \mathcal{T}[v \mapsto t]; \mathrm{b}_i$ for $i = 0, 1$,
- $\mathcal{T}\left[v \mapsto \left\{ \begin{array}{ccc} \mathrm{b}_0(v_0) & \mapsto & t_0 \\ \mathrm{b}_1(v_1) & \mapsto & t_1 \end{array} \right\}(t)\right] = \langle \mathcal{T}[v \mapsto t], 1 \rangle; \mathrm{case}\{\mathcal{T}[(v_0, v) \mapsto t_0], \mathcal{T}[(v_1, v) \mapsto t_1]\}$,
- $\mathcal{T}[v \mapsto c_i(t)] = \mathcal{T}[v \mapsto t]; c_i$,
- $\mathcal{T}\left[v \mapsto \left\{ \begin{array}{ccc} \mathrm{c}_1(v_1) & \mapsto & t_1 \\ & \vdots & \\ \mathrm{c}_n(v_n) & \mapsto & t_n \end{array} \right\}(t)\right] = \langle \mathcal{T}[v \mapsto t], 1 \rangle; \mathrm{case}^L\{\mathcal{T}[(v_1, v) \mapsto t_1], ..., \mathcal{T}[(v_n, v) \mapsto t_n]\}$,
- $\mathcal{T}\left[v \mapsto \left\{\middle| \begin{array}{ccc} \mathrm{c}_1 : v_1 & \mapsto & t_1 \\ & \vdots & \\ \mathrm{c}_n : v_n & \mapsto & t_n \end{array} \middle|\right\}(t)\right] = \langle \mathcal{T}[v \mapsto t], 1 \rangle; \mathrm{fold}^L\{\mathcal{T}[(v_1, v) \mapsto t_1], ..., \mathcal{T}[(v_n, v) \mapsto t_n]\}$,
- $\mathcal{T}\left[v \mapsto L\left\{ \begin{array}{ccc} v_1 & \mapsto & t_1 \\ & \vdots & \\ v_m & \mapsto & t_m \end{array} \right\}(t)\right] = \langle \mathcal{T}[v \mapsto t], 1 \rangle; \mathrm{map}^L\{\mathcal{T}[(v_1, v) \mapsto t_1], ..., \mathcal{T}[(v_m, v) \mapsto t_m]\}$.
- $\mathcal{T}\left[v \mapsto \left(\middle| w \mapsto \begin{array}{c} \mathrm{d}_1 : t_1 \\ \vdots \\ \mathrm{d}_m : t_m \end{array} \middle|\right)(t)\right] = \langle \mathcal{T}[v \mapsto t], 1 \rangle; \mathrm{unfold}^L\{\mathcal{T}[(v, w) \mapsto t_1], ..., \mathcal{T}[(v, w) \mapsto t_n]\}$
- $\mathcal{T}\left[v \mapsto \left(\begin{array}{c} \mathrm{d}_1 : t_1 \\ \vdots \\ \mathrm{d}_m : t_m \end{array} \right)\right] = \mathrm{record}^L\{\mathcal{T}[v \mapsto t_1], ..., \mathcal{T}[v \mapsto t_n]\}$

8 Acknowledgements

Robin Cockett would like to acknowledge the financial support of the ARC and intellectual support of the Sydney category seminar for which he was a research fellow from January 1990 to June 1991 working under Ross Street of Macquarie University. Many of the ideas underlying charity were developed while he was in Sydney.

Tom Fukushima would like to acknowledge the support of the Formal Methods Group at the University of Calgary and NSERC who allowed and funded his trip to Macquarie University, Sydney, Australia so that the first version of the charity programming system could be developed with Robin Cockett, and Macquarie University who provided monetary support and facilities.

References

[AVA74] J. D. Ullman A. V. Aho, J. E. Hopcroft. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

[Bur87] R. Burstall. Inductively defined functions in functional programming languages. Laboratory for foundations of computer science, University of Edinburgh, 1987.

[BW85] M. Barrs and C. Wells. *Toposes, Triples and Theories.* Springer-Verlag, New York Berlin Heidelberg Tokyo, 1985.

[CC] J. R. B. Cockett and H. G. Chen. Categorical combinators. Submitted to 3rd Banff Higher-order Workshop.

[Coc90] J. R. B. Cockett. List-arithmetic distributive categories; locoi. *J. Pure Appl. Algebra*, 66:1–29, 1990.

[Coc91a] J. R. B. Cockett. The fundamental theorem of data structures for a locos (draft manuscript). Macquarie Univ., May 1991.

[Coc91b] J. R. B. Cockett. Notes on strength, datatypes, and shape (draft manuscript). Macquarie Univ., May 1991.

[CS92] J. R. B. Cockett and D. Spencer. Strong categorical datatypes II. Submitted, 1992.

[EMP91] M. Fokkinga E. Meijer and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.

[Hag87] T. Hagino. *A Categorical Programming Language.* PhD thesis, University of Edinburgh, 1987.

[Kel] G. M. Kelly. On clubs and data-type constructors. University of Sydney.

[Kel82] G. M. Kelly. Structures defined by finite limits in the enriched context i. *Cahiers de Top. et Géom. Différentielle*, 23:3–42, 1982.

[Lam69] J. Lambek. *Deductive systems and categories II: Standard constructions and closed categories*, volume 86 of *Lecture notes in Mathematics*, pages 76–122. Springer–Verlag, 1969.

[Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proc. IEEE Conf. on Logic in Computer Science*, pages 14–23, 1989.

[Tur79a] D. A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44(2):67–270, 1979.

[Tur79b] D. A. Turner. A new implementation technique for applicative languages. *Software — Practice and Experience*, 9:31–49, 1979.

[Wad89] Philip Wadler. Theorems for free! In *The Fourth Internaltional Conferece on Functional Programming Languages and Computer Architecture*, pages 347–359. Addison-Wesley, 1989.

[Wad90] P. Wadler. Comprehending monads. *ACM Conference on Lisp and Functional Programming*, June 1990.

[Wal91] R. F. C. Walters. *Categories and computer science.* Carslaw Publications, Sydney, 1991.

[Wra89] G. C. Wraith. A note on categorical datatypes. In *Category theory and computer science*, volume 389 of *Lecture Notes in Computer Science*, pages 118–127. Springer-Verlag, 1989.

A Charity programming system

The notation used in this document unfortunately does not translate directly into ascii. Therefore we summarize the differences between the mathematical notation above and the charityprogramming system's notation. For a complete description of the charity programming system of which only a part of is described here, we refer the reader to the Charity User's Manual. Spacing is not important in the following examples. All system commands must be followed by a "." (period).

Both the charity programming system and the Charity User's Manual are available via anonymous ftp from cpsc.ucalgary.ca in pub/charity.

A.1 Datatype definition

The differences in the datatype definition are that the "$\longrightarrow$" is replaced by "`->`", and the "$\times$" (product) is replaced by "*" (asterisk) for type expressions.

$$\begin{array}{ll} \texttt{data} & L(A)\ \texttt{->}\ C\ \texttt{=} \\ & c_1\texttt{:}\ E_1(A\texttt{,}C)\ \texttt{->}\ C \\ & \vdots \\ \texttt{|} & c_n\texttt{:}\ E_n(A\texttt{,}C)\ \texttt{->}C. \end{array}$$

For example, the definition for btree would be

```
data btree(A,B) -> C =
        leaf : A -> C
      | node : (C * (B * C)) -> C.
```

The definitions for natural numbers and booleans are predefined in the system as

```
data nat -> C = Z : 1 -> C | S : nat -> C.
data bool -> C = true : 1 -> C | false : 1 -> C.
```

where `Z` is zero and `S` is the successor.

A.2 Terms

To define the map

$$k\{f_1, ..., f_n\} : X \longrightarrow Y; v \mapsto t$$

to the system, one would write:

$$\texttt{def}\ k\{f_1, ..., f_n\}(v) = t.$$

and the notational changes for terms are:

- "$\mapsto$" (maps to) is written "`=>`" (equals, greater than).
- The case operation

$$\left\{ \begin{array}{ccc} c_1(v_1) & \mapsto & t_1 \\ \vdots & & \\ c_n(v_n) & \mapsto & t_2 \end{array} \right\}(t) \qquad \text{is written} \qquad \begin{array}{ll} \{ & c_1(v_1)\ \texttt{=>}\ t_1 \\ & \vdots \\ \texttt{|} & c_n(v_n)\ \texttt{=>}\ t_n \\ \} & (t) \end{array}$$

- The fold operation

$$\left\{\!\!\left|\begin{array}{ccc} c_1 : patt_1 & \mapsto & expr_1 \\ & \vdots & \\ c_m : patt_m & \mapsto & expr_m \end{array}\right|\!\!\right\}(expr) \qquad \text{is written} \qquad \begin{array}{llll} \{| & c_1 : patt_1 & \texttt{=>} & expr_1 \\ | & c_2 : patt_2 & \texttt{=>} & expr_2 \\ & & \vdots & \\ | & c_m : patt_m & \texttt{=>} & expr_m \\ |\} & (expr) & & \end{array}$$

- The map operation

$$L\left\{\begin{array}{ccc} v_1 & \mapsto & t_1 \\ \vdots & & \\ v_m & \mapsto & t_m \end{array}\right\}(t) \qquad \text{is written} \qquad \begin{array}{lll} L & \{ & v_1 \texttt{ => } t_1 \\ & & \vdots \\ & , & v_m \texttt{ => } t_m \\ & \} & (t) \end{array}$$

- Abstraction is written:

$$\{v\texttt{=>}t'\}(t)$$

- The unfold operation

$$\left(\!\!\left|\texttt{patt} \mapsto \begin{array}{c} d_1 : expr_1 \\ \vdots \\ d_m : expr_m \end{array}\right|\!\!\right)(\texttt{expr}) \qquad \text{is written} \qquad \begin{array}{lll} (| & patt \texttt{ =>} & \\ & d_1 : & expr_1 \\ | & d_2 : & expr_2 \\ & \vdots & \\ | & d_m : & expr_m \\ |) & (expr) & \end{array}$$

- The record operation

$$\left(\begin{array}{c} d_1 : expr_1 \\ \vdots \\ d_m : expr_m \end{array}\right) \qquad \text{is written} \qquad \begin{array}{llll} (& d_1 & : & expr_1 \\ , & d_2 & : & expr_2 \\ & & \vdots & \\ , & d_m & : & expr_m \\ ) & & & \end{array}$$

For example, the definition for divide (section 4.4) would be:

```
def divide{p} (l) =
     { nil       () => leaf(nil)
     | cons(a,tail) =>
          { (l1,(a,l2)) => node(leaf(l1),(a,leaf(l2))) }
                            (split_list{p}(tail,a))
     } (l).
```

Note that the definition of divide uses the previously defined definition split_list. Recursive calls are not allowed.

The command `eval` is used to evaluate closed terms. For example:

```
eval divide{gt}([S(S(Z)),S(Z),S(S(S(Z))),Z]).
```

would apply divide to the list of numbers [2,1,3,0] with predicate greater than.

A.3 Combinator notation

Combinator notation is also available but has to be translated into ascii. The differences are:

- case^L is written case_L,
- fold^L is written fold_L,
- $\text{map}^L; L$ is written map_L.

The command used to define a combinator expression to the system under name *ident* is:

$$\texttt{cdef}\ ident\{f_1, ..., f_n\} = expr.$$

where *expr* is a combinator expression which uses maps $f_1, ..., f_n$.

The command `ceval` is used to evaluate closed combinator expressions. For example:

```
ceval < Z, Z ; S > ; p1.
```

extracts 1 from the pair (0,1).

A.4 Example

To give a flavor for programming in the charity programming system we give a short example session. In this example, we show how to define the natural numbers and some maps to use them.

We will use a shorthand for lists supplied by the system:

$$[x_1, x_2, ..., x_n] \equiv \text{cons}(x_1, \text{cons}(x_2, \dots \text{cons}(x_n, \text{nil})))$$

When charity is started, it will give a message signaling that it is ready. User commands can be entered at the "`>`" prompt and the "`+`" prompt signals that the command is being continued over several lines.

```
Charity ready
>
```

The first command required for this example is

```
> restart_base.
```

which erases all predefined datatypes and combinators, and thus allowing us to redefine our own natural numbers below.

We define the natural numbers by the command

```
> data mynat -> X =
+          zero : 1 -> X
+        | succ : X -> X.
```

which says that `mynat` is a type and has constructors `zero` and `succ` (successor). Thus values of type `mynat` may be created by applying `zero` to `()` (ie. something of type `1`) or by applying `succ` to an expression of type `mynat`. For convenience the `()` may be omitted. For example:

$$\begin{array}{rcl} \texttt{zero} & = & 0 \\ \texttt{succ(succ(zero))} & = & 2 \\ \texttt{succ(}n\texttt{)} & = & n+1 \end{array}$$

After the command has been entered, the system will display information about the combinators created (this information is mainly for programming at the combinator level as opposed to the term-logic level).

Next, we define addition using the fold operation:

```
> def myadd(x,y) =
+       {| zero: () => y
+        | succ: n => succ(n)
+        |}(x).
```

We can think of `x` as being a code sequence of a `zero` followed by `succ`s. The fold operation says that `zero` is replaced by `y` and then each `succ` is replaced by `succ`. For example, if `x` = `succ(succ(zero))` and `y` = `succ(zero)` then

x = `zero ; succ ; succ`
replacing the `zero` by `y` gives
$\Longrightarrow$ `succ(zero) ; succ ; succ`
then replacing each `succ` by `succ` gives
$\Longrightarrow$ `succ(zero) ; succ ; succ`
= `succ(succ(succ(zero)))` = 3

To add 2 to 1 in the system we type:

```
> eval myadd(succ(succ(zero)),succ(zero)).

typing : 'a -> mynat

succ(succ(succ(zero)))
```

the type of the expression as well as the result is displayed.

The case operation can be used for the predecessor map.

```
> def mypred(x) =
+       { zero () => 0
+       | succ(n) => n
+       }(x).
> eval mypred(succ(succ(zero))).

typing : 'a -> mynat

succ(zero)
```

This says that if `x` is `zero` then return `zero` otherwise `x` is the successor of a number `n` in which case we return `n`.

An example of the map operation is "map" which given `L` which is of type `list`, apply `f` to every element `x` in the list.

```
> def mymap{f} (L) = list{x => f(x)}(L).
```

Now to double every element in a list of natural numbers:

```
> def double(x) = myadd(x,x).
> eval mymap{double} ([zero, succ(zero), succ(succ(zero))]).

typing : 'a -> list(mynat)

[zero,succ(succ(zero)),succ(succ(succ(succ(zero))))]
```