J. Functional Programming **1** (1): 1–000, January 1998. Printed in the United Kingdom 1

Functorial ML

C.B. JAY
School of Computing Sciences, University of Technology, Sydney, P.O. Box 123 Broadway, 2007, Australia. cbj@socs.uts.edu.au.

G. BELLÈ, E. MOGGI*
Dipartimento di Informatica, Università di Genova, via Dodecaneso 35, 16146 Genova, Italy. gbelle,moggi@disi.unige.it.

Abstract

We present an extension of the Hindley-Milner type system that supports a generous class of type constructors called functors, and provide a parametrically polymorphic algorithm for their mapping, i.e. for applying a function to each datum appearing in a value of constructed type. The algorithm comes from shape theory, which provides a uniform method for locating data within a shape. The resulting system is Church-Rosser and strongly normalizing, and supports type inference. Several different semantics are possible, which affects the choice of constants in the language, and are used to illustrate the relationship to polytypic programming.

Capsule Review

A wide class of type constructors (functions producing types from types) used in functional programming are *functorial*, in the sense that they can be extended to mappings from functions to functions satisfying a few simple laws. The 'map' functional for lists is the prototypic example. Moreover, this additional structure for type constructors is very useful for expressing properties of some recursively defined datatypes and (hence) in the construction of algorithms expressed in the functional style. The work of the 'Dutch-Oxford' school of functional programming and algorithm design provides ample evidence of the utility of the functorial approach. So one is naturally led to the idea of 'functorial', rather than merely functional, programming.

Now amongst the many different functorial actions that a particular type constructor might possess, there is usually a 'canonical' one. Rather than have the programmer supply the definition of the functorial action explicitly (which would rapidly obscure the structure of one's functorial program) it would be very nice if one could write 'functorially polymorphic' programs in which the canonical functorial action is inferred automatically from the syntax of the type constructor. Since the algorithms for functorial actions differs greatly from one type constructor to another, such functorial polymorphism is rather subtle compared with the usual form of polymorphism found in most typed functional languages. This paper presents one way of achieving this new form of functor-based polymorphism.

* This research was partially supported by MURST and ESPRIT WG AppSem.

1 Introduction

The interplay between type theory, programming language semantics and category theory is now well established. Two of the strongest examples of this interaction are the representation of function types as exponential objects in a cartesian closed category (Lambek & Scott, 1986) and the description of polymorphic terms as natural transformations (e.g. (Bainbridge *et al.*, 1990)). For example, the operation of appending lists can be represented as a natural transformation $L \times L \Rightarrow L$ where $L : \mathcal{D} \to \mathcal{D}$ is the list functor on some category $\mathcal{D}$. Of course, these natural transformations must have associated functors for their domain and codomain. System F supports a notion of *expressible functor*, i.e. a type constructor and a corresponding action on functions (Reynolds & Plotkin, 1990), but such encodings are rather unsatisfactory (Section 15.1.1). In particular, the action of a functor on morphisms, its *mapping*, must be defined anew for each choice of type constructor.

A better approach, primarily advocated by adherents of the Bird-Meertens style (e.g. (Meijer *et al.*, 1991; Meijer & Hutton, 1995; Jeuring, 1995)), is to give a combinator for mapping, whose type can be expressed as:

$$\mathrm{map} : \forall F : 1.\forall X, Y.(X \to Y) \to FX \to FY.$$

That is, for any functor $F : 1$ (i.e. taking one argument), types X and Y, and any morphism $f : X \to Y$ we have

$$\mathrm{map}\ F\ X\ Y\ f : FX \to FY$$

whose action is to take each datum of type X in FX and apply f to it. Unfortunately, the existence of this type does not solve the problem of realizing this high-level algorithm, as there remains the question of how to *find* the data.

Naturally, one can use the functor to determine the algorithm. There are basically two ways to do this. One method is to have the user specify the mapping algorithm, say by instantiating a *constructor class* (Jones, 1995). In this particular case, the functor and type arguments are suppressed to obtain map f a, since the choice of functor can be inferred from the type of a. Unfortunately, it follows that if F and G are functors such that FX and GY are intended to be the same for some types X and Y then a dummy constructor must be introduced to distinguish them.

The other method is to generate the mapping algorithm automatically, from the structure of the functor. This results in a small loss of flexibility, but saves the user from supplying repetitious algorithms. Charity (Cockett & Fukushima, 1992) encodes this directly. Jeuring (Jeuring, 1995) uses a pre-processor to determine the appropriate Haskell code for mapping and other *polytypic* operations. *Intensional polymorphism* (Harper & Morrisett, 1995) is a general technique for describing type-dependent operations in an extension of ML, designed to obtain more efficient compilation. Although in the same spirit as the other approaches, the lack of sum types makes it hard to make direct comparisons.

Perhaps surprisingly, there is a generous class of covariant functors for which it is possible to describe a mapping algorithm that is independent of the choice of functor, i.e. which support *parametric functorial polymorphism.* The first such algorithms for (polymorphic folding) were produced for a small experimental language **P2** (Jay, 1995a). Polymorphic mapping for the *covariant* type system is developed in (Jay, 1997). This paper presents an extension of Hindley-Milner called Functorial ML, or FML, which supports parametric functorial polymorphism.

For example, it supports:

$$\begin{array}{rcl}\mathrm{map}\ f\ (\mathrm{cons}\ h\ t) & \to_* & \mathrm{cons}\ (f\ h)\ (\mathrm{map}\ f\ t)\\ \mathrm{map}\ f\ (\mathrm{leaf}\ x) & \to_* & \mathrm{leaf}\ (f\ x)\end{array}$$

where cons is the usual list constructor, and leaf is the leaf constructor for binary trees with labeled leaves. It is important to note that these evaluations are not achieved by pattern matching on primitive constants, but that the constructors cons and leaf have internal structure, which is used in the reduction to find the data in a uniform way. We can also use mapping within a let-construct:

$$\mathrm{let}\ g = \mathrm{map}\ f\ \mathrm{in}\ \langle g\ (\mathrm{cons}\ h\ t), g\ (\mathrm{leaf}\ x)\rangle$$

where $\langle -, - \rangle$ is the pairing for binary products. The polymorphic mapping allows us to support polymorphic folding, too, as will be shown in the body of the paper. Functors of many variables are also catered for. For example, we have

$$\mathrm{map}_2\ f\ g\ (\mathrm{in}_0\ t) \ \to\ \mathrm{in}_0\ (f\ t) \tag{1}$$

$$\mathrm{map}_2\ f\ g\ (\mathrm{pair}\ s\ t) \ \to\ \mathrm{pair}\ (f\ s)\ (g\ t). \tag{2}$$

where in_0 is the left inclusion to a binary sum, Thus, $\mathrm{map}_2\ f\ g$ is equally able to act on values whose type is a sum or product, etc.

Shape theory (Jay, 1995b) provides the basis for these algorithms. It is a new approach to data types based on the idea of decomposing values into their shape (or data *structure*) and the data which is stored within them. The data structure corresponds to the type constructor, or functor, whose argument is the type of the data. Thus *shape polymorphism* is closely linked to functorial polymorphism, as distinct from the *data polymorphism* of operations like append. Thus, the data-shape decomposition supports uniform mechanisms for storing data within a shape, which are exploited by our mapping algorithm.

To see how this works, consider the projection functors $\Pi_i^m : \mathcal{D}^m \to \mathcal{D}$ which pick out the ith argument from m. It is tempting to identify $\Pi_0^2(X, X)$ and $\Pi_1^2(X, X)$ with X but this would obliterate the shape-data distinction. Rather, these types are "isomorphic", a situation captured by terms

$$\mathrm{tag}_{2,j} : X_j \to \Pi_j^2(X_0, X_1)$$

(for $j \in 2$) and their inverses. Now given $x : X$ we have the reductions:

$$\mathrm{map}_2\ f_0\ f_1\ (\mathrm{tag}_{2,j}\ x) \to \mathrm{tag}_{2,j}\ (f_j\ x).$$

In other words, the isomorphisms are used to determine where to find the data associated to each argument of the functor. Similarly, we have isomorphisms to disambiguate composite functors, e.g.

$$\mathrm{pack}_{1,1} : F(G(X)) \to F\langle G\rangle^1(X).$$

Its source is the functor F applied to $G(X)$ while its target is the composite functor $F\langle G\rangle^1$ applied to X. The corresponding reduction for map is:

$$\text{map } f \ (\text{pack}_{1,1} \ t) \to \text{pack}_{1,1} \ (\text{map } (\text{map } f) \ t)$$

whose corresponding diagram is:

$$\begin{tikzcd} F(G(X)) \arrow[r, "\text{pack}_{1,1}"] \arrow[d, "\text{map (map } f)"'] & F\langle G\rangle^1(X) \arrow[d, "\text{map } f"] \\ F(G(Y)) \arrow[r, "\text{pack}_{1,1}"'] & F\langle G\rangle^1(Y). \end{tikzcd}$$

These isomorphisms may be viewed as a systematic method for resolving the ambiguities otherwise addressed by the dummy constructors mentioned above. The other approaches use implicit substitution to handle functor composition, making a uniform algorithm impossible.

The significance of these isomorphisms becomes particularly clear in certain application contexts. For instance, in distributed or parallel computing the shape-data distinction can be used to describe data distributions (Jay, 1995c) in which case the isomorphisms represent redistributions of the data. Also, such isomorphisms between different composites are central to bicategories (Benabou, 1967).

The polymorphism of the mapping algorithm can be captured in a system that supports inference of functors as well as types. We work with an extension of the Hindley-Milner types which supports a syntactic class of functors, as well as those of types and type schema. Another possibility is to identify types and type schema (so extending system F with functors). Such a system is used in the proof of strong normalization. Again, one could identify types with functors of arity 0, at the cost of introducing another pair of isomorphisms. We emphasize the tri-partite division for reasons of clarity, and to obtain type inference.

In this paper we will focus on FML with untyped terms, i.e. à la Curry (in the terminology of (Barendregt, 1992)), to emphasize parametric functorial polymorphism. However, FML à la Church is very important, too: it allows a more aggressive use of type information (as advocated in (Harper & Morrisett, 1995)) and it is more suitable for a semantic investigation.

The paper is structured as follows. This section, Section 1 introduces the paper. Then Section 2 introduces the functors, their syntax and semantics. Section 3 introduces the basic term structure and proves elementary properties, relative to a set of term constants. Section 4 considers a variety of functor-polymorphic constants motivated by the desired choice of functor semantics. It also introduces some syntactic sugar for monads. This is used in Section 5 where shape polymorphic programs for matching and unification are constructed.

Section 6 looks at extensions to the basic type system. As well as considering quantification by functor arities, it considers recursion over functor construction, which leads to polytypism. This can be used to define the various constants introduced earlier, provided that the semantics of functors is suitably constrained. Section 7 contrasts FML with **PolyP**, a polytypic language currently under development (Jansson & Jeuring, 1997). Section 8 concludes the body of paper.

2 Functors, types and type schema

2.1 Functors

Only elementary concepts of category theory will be required in this paper, as found in almost any reference work, e.g. (MacLane, 1971; Barr & Wells, 1990). Consider a fixed category, $\mathcal{D}$. A *(covariant) functor* is a structure-preserving morphism of categories, i.e. it maps objects to objects and morphisms to morphisms, so as to preserve the sources and targets of the morphisms, the composition and identities. The identity arrow on an object A may be written id_A. The symbols m and n will denote natural numbers throughout this paper. The category $\mathcal{D}^m$ has objects given by m-tuples of objects of $\mathcal{D}$ and arrows given m-tuples of arrows of $\mathcal{D}$ with the obvious point-wise composition and identities. A functor $F : \mathcal{D}^m \to \mathcal{D}$ is said to be a *functor on $\mathcal{D}$ of arity m* which may be written as $F : m$ (as $\mathcal{D}$ is understood from the context). The action of such a functor on a sequence of arrows $f_i : X_i \to Y_i$ for $i \in m$ will be written as

$$\mathrm{map}_m\ f_0\ \ldots f_{m-1} : F(X_0, \ldots, X_{m-1}) \to F(Y_0, \ldots Y_{m-1})\ .$$

Here are some elementary examples and constructions. The binary functors $+, \times : 2$ represent sums and products, and $1 : 0$ is the functor of no arguments that produces the terminal object, corresponding to the unit type. Let i range over $m = \{0, \ldots, m-1\}$, also written $i : m$. The ith *projection functor* of m arguments is $\Pi_i^m : m$. A sequence $G_i : n$ of functors may be written as $G_{i:m}$ or even $\overline{G}$ when the choice of m is either clear from the context, or irrelevant. Similar notation will be used for other sequences below, of types, etc. If $F : m$ then

$$\mathcal{D}^n \xrightarrow{\langle \overline{G} \rangle} \mathcal{D}^m \xrightarrow{F} \mathcal{D}$$

written $F\langle\overline{G}\rangle$ is their *composite.* If $F : m+1$ is a functor then $\mu F : m$ is its *initial algebra functor* which represents the functor whose action on the tuple $\overline{X}$ yields the initial $F(\overline{X}, -)$-algebra with action

$$\mathrm{wrap}_m : F(\overline{X}, \mu F(\overline{X})) \to \mu F(\overline{X})$$

which gives a minimal solution of the domain equation $F(\overline{X}, Y) \cong Y$.

Functoriality is fundamental to the interpretation of the algebras whose initiality we are considering. For example, given $F : 1$ and an algebra $f : F(Y) \to Y$ the algebra homomorphism $(\mathrm{fold}_0\ f) : \mu F \to Y$ makes the following square commute:

$$\begin{tikzcd} F(\mu F) \arrow[r, "\mathrm{wrap}_0"] \arrow[d, "\mathrm{map}_1\ (\mathrm{fold}_0\ f)"'] & \mu F \arrow[d, "\mathrm{fold}_0\ f"] \\ F(Y) \arrow[r, "f"'] & Y \end{tikzcd}$$

This is the basis for the corresponding term reduction

$$\mathrm{fold}_0\ f\ (\mathrm{wrap}_0\ t) > f\ (\mathrm{map}_1(\mathrm{fold}_0\ f)t)$$

which shows that $\mathrm{fold}_0\ f$ acts by recursively mapping itself across all of the substructures of $\mathrm{wrap}_0\ t$ and then applying f to the result.

The usual presentations of fold_0 expand this definition for each choice of functor, typically by a case analysis on the functor constructors.

Now consider initial algebras for $F : 2$. Each $f : X \to Y$ determines an algebra

$$\mathrm{wrap}_1 \circ (\mathrm{map}_2\ f\ \mathrm{id}) : F(X, \mu F(Y)) \to \mu F(Y)$$

whose corresponding homomorphism $\mu F(X) \to \mu F(Y)$ defines the functoriality of μF. Thus, the mapping on $F : 2$ defines the initial algebra structure on $\mu F(X)$ which in turn supports the functoriality of μF. Thus, despite the delicate interplay between mapping and folding, it is clear that mapping, i.e. functoriality, is the the fundamental concept, and further, that it cannot be captured by the purely type-based notions of introduction and elimination rules. It also emphasizes the need to consider all functors together, rather than separating them according into arities, as occurred in the earlier work of one of us (Jay, 1995a) or in the study of polytypism, e.g. (Jansson & Jeuring, 1997).

The class of functors obtained by closing up under the constructions above have been widely studied in the context of initial algebra theory, starting with (Goguen *et al.*, 1977) and more recently as the *regular functors* of (Meertens, 1996).

2.2 Syntax

These constructions motivate the choice of functors in the following description of the raw syntax for *functors* F, *types* τ and *type schema* σ in FML:

$$\begin{array}{rcl} F, G & ::= & X \mid C \mid \Pi_i^m \mid F\langle\overline{G}\rangle^n \mid \mu F \\ \tau & ::= & X \mid F(\overline{\tau}) \mid \tau_1 \to \tau_2 \\ \sigma & ::= & \tau \mid \forall X : \mathrm{T}.\sigma \mid \forall X : m.\sigma. \end{array}$$

In addition to the constructions above, functors include variables X and constant functors C. Types are variables, functor applications, and function types. The application of a functor to a tuple $\tau_{i:m}$ of types represents the action of the categorical functor on objects. Note that the usual ancillary type constructors, such as products and sums, have been treated as functors. The key exception is the function type constructor, which is contravariant in its first argument.

Type schema include the types, and are closed under universal quantification over type variables and over functors of given arity. The former quantification is familiar from Hindley-Milner and is used to express data polymorphism; the latter quantification will allow us to express functorial polymorphism.

Notation 2.1 We adopt the following notational conventions throughout the paper. X and Y range over functor and type variables. C ranges over functor constants, in particular $+$, $\times$ and 1. The symbols F and G range over functors (though sometimes are used as functor variables), τ ranges over types, and σ ranges over type schema. A type τ may be distinguished from functors or type schema by giving it the fixed arity $\tau : \mathrm{T}$. We write $\tau_1 \times \tau_2$ for $\times(\tau_1, \tau_2)$ and $\tau_1 + \tau_2$ for $+\langle\tau_1, \tau_2\rangle$ and 1 for the type $1()$. Composition binds tighter than the initial algebra constructor μ.

A *type context* Δ is a sequence of type variables with assigned arities (either $X : n$ or $X : \mathrm{T}$) with no repetition of variables. We may identify Δ with a partial function from functor and type variables to arities, and write $\mathrm{DV}(\Delta)$ for its domain. Also, each functor constant C has an associated arity n_C fixed upon its introduction.

For each of the syntactic categories above we define corresponding judgments for asserting well-formedness as follows:

- $\Delta \vdash$ means that Δ is a well-formed typed context.
- $\Delta \vdash F : n$ means that F is a functor of arity n in context Δ.
- $\Delta \vdash \tau : \mathrm{T}$ means that τ is a type in context Δ.
- $\Delta \vdash \sigma$ means that σ is a type schema in context Δ.

The corresponding inference rules are given in Figure 1 (where the symbol i ranges over m). The formation rules for functors express the constraints on arities implicit in the category theory. In general, we may use the symbol J to represent judgments.

The *free variables* of a functor are defined in the usual way, and the functors are defined to be equivalence classes of well-formed functor expressions under α-conversion. Types and schema are defined similarly. The quantification $\forall\Delta.\sigma$ of a type scheme σ by a type context Δ is defined as follows:

$$\begin{array}{rcl} \forall.\,\sigma & = & \sigma \\ \forall(\Delta', X : n).\,\sigma & = & \forall\Delta'.(\forall X : n.\,\sigma) \\ \forall(\Delta', X : \mathrm{T}).\,\sigma & = & \forall\Delta'.(\forall X : \mathrm{T}.\,\sigma) \end{array}$$

Lemma 2.2

1. Uniqueness of derivation: each judgment $\Delta \vdash J$ has at most one derivation (up to α-conversion).
2. Uniqueness of arity: if $\Delta \vdash F : n_j$ is derivable for $j \in 2$ then $n_0 = n_1$.

Proof

For the first, use induction on the size of the derivation of $\Delta \vdash J$. For the second, use induction on the structure of F. $\square$

Contexts

$$(\text{empty})\ \frac{}{\emptyset \vdash} \qquad (\text{functor})\ \frac{\Delta \vdash}{\Delta, X : n \vdash}\ X \notin \mathrm{DV}(\Delta)$$

$$(\text{type})\ \frac{\Delta \vdash}{\Delta, X : \mathrm{T} \vdash}\ X \notin \mathrm{DV}(\Delta)$$

Functors

$$(X)\ \frac{\Delta \vdash}{\Delta \vdash X : m}\ m = \Delta(X) \qquad (C)\ \frac{\Delta \vdash}{\Delta \vdash C : n_C}$$

$$(\text{FPi})\ \frac{\Delta \vdash}{\Delta \vdash \Pi^m_i : m} \qquad (\text{Fcomp})\ \frac{\Delta \vdash F : m \quad \Delta \vdash G_0 : n\ \ldots\ \Delta \vdash G_{m-1} : n}{\Delta \vdash F\langle G_{i:m}\rangle^n : n}$$

$$(\text{Fmu})\ \frac{\Delta \vdash F : m+1}{\Delta \vdash \mu F : m}$$

Types

$$(X)\ \frac{\Delta \vdash}{\Delta \vdash X : \mathrm{T}}\ \mathrm{T} = \Delta(X) \quad (\text{Fapp})\ \frac{\Delta \vdash F : m \quad \Delta \vdash \tau_0 : \mathrm{T}\ \ldots\ \Delta \vdash \tau_{m-1} : \mathrm{T}}{\Delta \vdash F(\tau_{i:m}) : \mathrm{T}}$$

$$(\rightarrow)\ \frac{\Delta \vdash \tau_1 : \mathrm{T} \quad \Delta \vdash \tau_2 : \mathrm{T}}{\Delta \vdash \tau_1 \rightarrow \tau_2 : \mathrm{T}}$$

Schema

$$(\tau)\ \frac{\Delta \vdash \tau : \mathrm{T}}{\Delta \vdash \tau} \qquad (\forall_m)\ \frac{\Delta, Y : m \vdash \sigma\{Y/X\}}{\Delta \vdash \forall X : m.\sigma}\ Y \notin \mathrm{DV}(\Delta)$$

$$(\forall)\ \frac{\Delta, Y : \mathrm{T} \vdash \sigma\{Y/X\}}{\Delta \vdash \forall X : \mathrm{T}.\sigma}\ Y \notin \mathrm{DV}(\Delta)$$

Fig. 1. FML types

2.3 Examples

Many of the usual data type constructors can be presented as functors. For example, consider a datatype definition

$$\mathrm{Maybe}_d\ \tau = ok\ \tau \mid fail.$$

We can regard this as specifying the type $\tau + 1$ or, more formally, $+(\tau, 1())$ i.e. the plus functor applied to the two types τ and $1()$. Note that 1 is a functor of arity 0, not a type, and so must be applied to 0 arguments to obtain a type. To express this as a functor of τ requires several abstractions. First, the type τ must be replaced by $\mathrm{Id}(\tau)$ where Id is the *identity functor* Π^1_0. Then, to respect arities, the type $1()$ must also be presented as a functor of arity 1 that ignores its argument, namely

$1\langle\rangle^1(\tau)$. This analysis yields

$$+(\mathrm{Id}(\tau), 1\langle\rangle^1(\tau))$$

i.e. the functor $+$ applied to two types built from τ. Finally, we can replace this by a functor of one variable, namely

$$+\langle \mathrm{Id}, 1\langle\rangle^1\rangle^1(\tau)$$

or

$$\mathrm{Maybe}_F = +\langle \mathrm{Id}, 1\langle\rangle^1\rangle^1.$$

Note that $\mathrm{Maybe}_F(\tau)$ and $\tau + 1$ are not identical types. They are obtained by applying functors of different arities, namely 1 and 2 respectively. This distinction is fundamental since the meaning of terms below will typically depend on the separation of the functor from its arguments.

The list functor shows the use of recursion in functor definition. This functor corresponds to the datatype definition $L_d X = \mathit{nil} \mid \mathit{cons}\ X \times L_d X$. We must construct the functor that maps types X and Y to $1 + X \times Y$. This is

$$+\langle 1\langle\rangle^2, \times\rangle^2$$

Now the list functor L is obtained by constructing the corresponding initial algebra functor (which maps X to $\mu Y.1 + X \times Y$)

$$L = \mu + \langle 1\langle\rangle^2, \times\rangle^2$$

Similarly, the datatype $T_d X = \mathit{leaf}\ X \mid \mathit{node}\ T_d X \times T_d X$ of binary trees with labeled leaves has corresponding functor

$$T = \mu + \langle \Pi^2_0, \times\langle \Pi^2_1, \Pi^2_1\rangle^2\rangle^2$$

and the datatype $R_d X = \mathit{leaf}\ X \mid \mathit{node}\ L_d(R_d X)$ of rose trees (having finite lists of branches) has corresponding functor

$$R = \mu + \langle \Pi^2_0, L\langle \Pi^2_1\rangle^2\rangle^2$$

2.4 Substitutions and unification

A *substitution* is a partial function S from functor and type variables to expressions for functors and types, respectively. The notation $S : \Delta_1 \to \Delta_2$ means that the Δ_i are well-formed contexts, $\mathrm{DV}(\Delta_1)$ is included in the domain of S and for each $X \in \mathrm{DV}(\Delta_1)$ we have

$$\Delta_2 \vdash SX : \Delta_1(X).$$

The action of such an S extends homomorphically to any expression that is well-formed in context Δ_1 (including those to be defined below). If $R : \Delta_2 \to \Delta_3$ is another substitution then their *composite* substitution $R\ S$ has action given by $(R\ S)X = R(SX)$.

S is a *renaming* if it is an injective function from variables to variables. Then for each type context Δ we have another such Δ_S obtained by applying S homomorphically to Δ.

Lemma 2.3

1. Renaming: if S is a renaming then $\Delta \vdash J$ implies $\Delta_S \vdash S(J)$.
2. Thinning: $\Delta_1, \Delta_2 \vdash J$ implies $\Delta_1, X : a, \Delta_2 \vdash J$ provided $X \notin \mathrm{DV}(\Delta_1, \Delta_2)$ and a is either T or an arity m.
3. Substitution: if $S : \Delta_1 \to \Delta_2$ is a substitution then $\Delta_1 \vdash J$ implies $\Delta_2 \vdash S(J)$.

Proof

Each of the proofs is by induction on the derivation of the premise. □

Let $\Delta \vdash J_j : a$ be well-formed functors or types having the same arity a for $j \in 2$. A *unifier* for (Δ, J_0, J_1) is a pair (Δ', S) such that $S : \Delta \to \Delta'$ is a substitution and $S(J_0) = S(J_1)$. Their *most general unifier* $\mathcal{U}(\Delta, J_0, J_1)$ is a unifier (Δ', S) such that if (Δ'', S') is any other unifier for them then there is a substitution $R : \Delta' \to \Delta''$ such that $S'(X) = R\ S(X)$ for all $X \in \mathrm{DV}(\Delta)$.

Lemma 2.4

If (Δ, J_0, J_1) has a unifier then it has a most general unifier.

Proof

Standard. Note that the introduction of functors does not require higher order unification since, for example, $\Pi_i^m(\overline{X})$ and X_i do not have a unifier. □

2.5 Semantics

A completely general semantic treatment would axiomatize the assumptions made about the underlying category, starting from, say, a *locos* (Cockett, 1990) as used in developing the foundations of shape theory (Jay, 1995b). A successful treatment should allow the results to be extended to various Kleisli categories built over the base, to represent computational monads (Moggi, 1991; Wadler, 1993). Another possibility, not explored here, is to interpret FML-functors as morphisms in a bicategory (Benabou, 1967). However, one already gains significant insights from set-theoretic models. We will also introduce domain-theoretic models (i.e. types as ω-cpos) to address some issues obscured in the set-theoretic treatment.

The simpler interpretations are *extensional.* The interpretation of functor application to types will be by applying (semantic) functors to objects. Thus they identify $\Pi_i^m(\overline{X})$ with X_i and $F\langle\overline{G}\rangle^m(\overline{X})$ with $F(\overline{G(\overline{X})})$ so that the canonical isomorphisms become equalities. To interpret the quantification occurring in type schema one should, say, use a set theory with universes, and interpret types as small sets. However, we will focus only on the novel part of the semantics, i.e. the interpretation FML functors, and so may safely ignore size issues.

Other, *intensional* models are more cumbersome, but able to represent the isomorphisms explicitly, so that functor composition is associative only up to isomorphism. These intensional interpretations motivate the introduction of FML constants corresponding to mediating isomorphisms.

Having fixed the choice of category, and thus the interpretation of types, the basic question is then: which functors? It is relatively easy to identify classes of functors closed under composition, so that the the key semantic point is that they be closed under the formation of initial algebras. Various alternatives will be given.

2.5.1 Set-theoretic semantics

Let **Set** be the category of (small) sets and functions. The obvious interpretation of an FML functor of arity m is as a functor $F : \mathbf{Set}^m \to \mathbf{Set}$. But the interpretation of μF is problematic, since there are functors which do not have an initial algebra for cardinality reasons, e.g. the power set functor $P(X) = \{X' | X' \subseteq X\}$. One may avoid this problem by interpreting FML functors within a suitably restricted class of functors on **Set**. We consider three possible choices: ω-colimit preserving functors (where "ω-colimit" means "colimit of an ω-chain"); functors shapely over lists; and, regular functors.

ω-colimit preserving functors. In any cartesian closed category $\mathcal{D}$ with small coproducts and ω-colimits the class of functors on $\mathcal{D}$ preserving ω-colimits enjoys the following properties:

- closure under composition;
- if $F : m+1$ preserves ω-colimits, then the the initial algebra functor μF exists and preserves ω-colimits;
- if $F_i : m$ preserves ω-colimits for each $i \in I$, then their coproducts, given by the functor $F(\overline{X}) = \coprod_{i \in I} F_i(\overline{X})$ does too;
- constant functors, projection functors Π_i^m and the binary product functor $\times$ preserve ω-colimits.

Thus, the class of ω-colimit preserving functors on **Set** is suitable for interpreting FML-functors. An example of ω-colimit preserving functor (not definable in FML) is the finite power set functor $\mathcal{P}_f$, while non-examples include functors of the form $F(X) = X^B$ where B is infinite.

Functors shapely over lists. A *functor shapely over lists* is a functor $F : \mathbf{Set}^m \to \mathbf{Set}$ equipped with a cartesian natural transformation

$$\delta_{\overline{X}} : F(X_i)_{i:m} \to \prod_{i:m} LX_i$$

i.e a natural transformation whose defining commuting squares are all pullbacks. Such a functor is determined, up to isomorphism, by its *object of shapes* $S = F(\overline{1})$ and its *arity* $E = \delta_{\overline{1}} : S \to N^m$ (where $N = L1$ is the natural numbers object) which specifies how much data of each type is required for each shape. An alternative definition of functor shapely over lists requires a cartesian natural transformation

$$\delta_{\overline{X}} : F(X_i)_{i:m} \to L(\coprod_{i:m} X_i)$$

The two definitions coincide when $m = 1$, and are *interchangable* for all purposes. In the rest of the paper we stick with the first definition.

In any locos, the class of functors shapely over lists is closed under the construction of initial algebra functors, and so is suitable for interpreting FML-functors (Jay, 1995b). Furthermore, functors shapely over lists preserve all pullbacks (since the list functor does), in particular the pullback

$$\begin{tikzcd} A \times B \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d] \\ B \arrow[r] & 1 \end{tikzcd}$$

that defines products. Thus one has an isomorphism called zip from the canonical pullback $FA \times_{\#} FB$ of the corresponding cospan into $F1$ to $F(A \times B)$. It takes a pair of values of the same shape and produces a shape filled with pairs. Therefore, this interpretation is useful to validate the extension of FML with shapely constants.

An example of functor shapely over lists is the matrix functor where $\mathrm{Matrix}(A)$ represents the matrices with entries of type A. Here δ produces a list of the entries, say, row by row. If we identify $\mathrm{Matrix}(1)$ with $N \times N$ then the arity function is simply multiplication. The list functor preserves all ω-colimits. In **Set** such colimits are preserved by pulling back, so that all functors shapely over list have this property, too. That these functors form a proper subclass of the ω-colimit preserving functors is illustrated by $\mathcal{P}_f$ whose object of shapes $\mathcal{P}_f(1) = 2$ is too small to represent all possible shapes of finite sets.

Inductive functors. The formal grammar for FML functor expressions (without functor variables) determines a class of functors, which we close under functor isomorphisms (to avoid making canonical choices) to obtain the *inductive functors*. The properties of this class have been studied by various authors interested in initial algebras, most recently by those investigating polytypism. Meertens (Meertens, 1996) dubbed them the *regular functors*, though: this name has been used for a slightly larger class elsewhere in the community (Jeuring & Jansson, 1996); it may clash with its use in the study of functors on regular categories; and, it contradicts the convention that trees may be thought of as irregular arrays.

A simple inductive argument shows that all inductive functors are shapely over lists. The converse fails since there are only countably many inductive functors (up to isomorphism) whereas the functors shapely over lists are indexed by the uncountable collection of functions into (powers of) N. For example, the matrix functor does not appear to be inductive.

Representations of functors shapely over lists. Since coproducts in **Set** are stable under pullbacks, functors shapely over lists can be decomposed into dependent sums of products, with one summand per shape:

$$F(\overline{X}) = \coprod_{s:S} (\prod_{i:m} X_i^{E(s,i)})$$

which can be represented by the pair $[\![F]\!] = \langle S : \mathbf{Set}, E : S \to N^m \rangle$. This explicit separation of the shape S from the data $\overline{X}$ provides a more intensional semantics of FML functors, which we now present in detail.

- $[\![\Pi_i^m]\!]$: $S = 1$ and $E(0, j) = (\text{if } i = j \text{ then } 1 \text{ else } 0)$

- $[\![F\langle\overline{G}\rangle^n]\!]$: $S = \coprod_{s:S_F}(\prod_{i:m} S_{G_i}^{E_F(s,i)})$ and $E\ (\langle s,f\rangle,j) = \coprod_{i:m}(\coprod_{e:E_F(s,i)} E_{G_i}(f\ i\ e,j))$ where $[\![F]\!] = \langle S_F, E_F\rangle$ and $[\![G_i]\!] = \langle S_{G_i}, E_{G_i}\rangle$
- μF: $S = \mu S. \coprod_{s:S_F} S^{E_F(s,m)}$ and $E(\langle s,f\rangle,i) = E_F(s,i) + \coprod_{e:E_F(s,m)} E(f\ e,i)$ where $[\![F]\!] = \langle S_F, E_F\rangle$. Here S can be viewed as the set of finite trees with nodes labeled by elements of S_F and branching determined by labels, while $E(\_,i)$ is a weight function (defined by induction on the structure of finite trees) such that a node labeled by s contributes $E_F(s,i)$ to the weight of the whole tree.

This interpretation is useful to motivate the introduction of canonical isomorphisms in FML. In fact, in this interpretation $\Pi_i^m(\overline{X}) \neq X_i$ and $F\langle\overline{G}\rangle^m(\overline{X}) \neq F(\overline{G(\overline{X})})$, but they are canonically isomorphic.

There is an alternative representation of functors shapely over lists as formal power series

$$F(\overline{X}) = \coprod_{n:N^m} C(n) \times \prod_{i:m} X_i^{n(i)}$$

for some $C : N^m \to \mathbf{Set}$ that establishes a link to Joyal's theory of species (Joyal, 1981).

Closed term model. Another intensional semantics is to interpret FML functors in the algebra of closed functor expressions, which can be seen as syntactic representations of inductive functors. This algebra is simply the closed term algebra for the following many-sorted algebraic signature

- Ω_m sort, for each $m \in N$
- $1 : \Omega_0$
- $\times, + : \Omega_2$
- $\Pi_i^m : \Omega_m$, for each $m \in N$ and $i \in m$
- $\circ_{m,n} : \Omega_m, \Omega_n^m \to \Omega_n$, for each $m, n \in N$
- $\mu_m : \Omega_{m+1} \to \Omega_m$, for each $m \in N$.

Here $\circ$ is used to represent composition. This interpretation can be used to support case analysis (or primitive recursion) on functor expressions, as discussed in Section 7 below.

2.5.2 Domain-theoretic semantics

Now let us consider semantics in the category $\mathbf{Cpo}$ of ω-cpos and ω-continuous functions. While many features are shared with the set-theoretic semantics, the domain-theoretic models expose some issues that are relevant for both the general semantics, and the implementation of FML. First, terms should be interpreted by continuous functions. In particular, the interpretation of the mapping combinator should be continuous. Therefore, the interpretation of an FML functor of arity m should be a *strong functor* $F : \mathbf{Cpo}^m \to \mathbf{Cpo}$, i.e. the action of F on morphisms must be *internalizable*, which in the case of $\mathbf{Cpo}$ means ω-continuous. Note that in

Set every functor is strong, while in **Cpo** there is at most one way of internalizing the action of a functor on morphisms (but this is not true in general).

Second, we *could* require that F preserve decidable objects, i.e. objects equipped with an equality arrow $\mathrm{eq} : X \times X \to 1+1$ making the following diagram a pullback:

$$\begin{tikzcd}
X \times X \arrow[r, "\langle \mathrm{id}, \mathrm{id} \rangle"] \arrow[d] \arrow[dr, phantom, very near start, "\lrcorner"] & X \times X \arrow[d, dashed, "eq"] \\
1 \arrow[r, "true"'] & 1+1.
\end{tikzcd}$$

In particular, this implies that $F(\overline{1})$ is decidable, so that equality of shapes can be tested. In **Set** every object has decidable equality, while in **Cpo** the objects with decidable equality are exactly the flat cpos (i.e. those cpos where the partial order coincides with the equality). They form a full reflective sub-category of **Cpo** isomorphic to **Set**.

As above, the interpretation of initial algebras requires consideration of restricted classes of (strong) functors, to which we now turn.

ω-colimit preserving strong functors. Let $F : m$ be a functor that preserves ω-colimits. If F is strong (respectively, preserves decidable objects) then so is its initial algebra functor. Thus, the strong functors, and their sub-class of functors preserving decidable objects, both form a suitable class for interpreting FML-functors. Among the strong functors which preserve ω-colimits but not flat cpos there are: the constant functors corresponding to non-flat cpos, and the lifting functor $FX = X_\perp$.

Functors shapely over lists. The functors shapely over lists are always strong, and so form models as before. Such a functor F preserves decidable objects iff $F(\overline{1})$ is decidable.

Representations of functors shapely over lists. Let F be shapely over lists in **Cpo**. Since list objects decompose into countable many disjoint components, one for each possible length, the same is true for F. It follows that the lifting functor is not shapely over lists. More generally, F's object of shapes S can be decomposed into a set U of connected components, i.e. represented as the colimit of a diagram $C : U \to \mathbf{Cpo}$. Thus, F can be represented (up to natural isomorphism) by

$$F(\overline{X}) = \coprod_{u:U} C(u) \times (\prod_{i:m} X_i^{E(u,i)})$$

for some set U, and $E : U \to N^m$. In these terms, preservation of decidable objects amounts to requiring that each connected component be a singleton, i.e. $C(u) = 1$ for all u. Therefore, these functors have the same representations adopted for functors on **Set** shapely over lists. Then the intensional semantics in **Cpo** can be taken verbatim from **Set**, the only difference being that a pair $\langle S, E : S \to N^m \rangle$

now represents a strong functor on **Cpo** extending the functor on **Set** with the same representation.

Inductive functors and closed term models These interpretations follow the same pattern as in **Set**.

3 Basic terms

This section introduces the basic term structure of FML, delaying the study of the combinators until the next section. As it is the novel combinators that give FML its flavor, this section is rather routine, but this makes the proofs routine, too. Further, all results are stable under change of the set of constants, both for functors and terms.

The terms are presented here à la Curry, i.e. with no explicit type or functor information. Explicitly typed terms, à la Church, will be considered in Section 6. There are two ways of assigning types to terms in the Hindley-Milner type system, either type schema or types (Tofte, 1988). The former has separate rules for abstracting and instantiating type variables, whereas the latter combines these with the rules for typing variables, combinators and the let-construct. We will present both versions, show them equivalent (Lemma 3.5), and then present a modified version of Milner's algorithm W for inferring types (see (Milner, 1978; Tofte, 1988)).

3.1 Syntax

The raw syntax for FML terms à la Curry is the same as that for the Hindley-Milner type system:

$$t ::= x \mid c \mid \lambda x.t \mid t_1\ t_2 \mid \text{let } x = t_1 \text{ in } t_2$$

Each combinator c has an associated type schema σ_c which will be define in the following section.

A *term context* is a sequence of $x : \sigma$ with no repetitions of term variables x. Define $\mathrm{DV}(\Gamma)$ to be the set of free term variables in Γ. A *context* $\Delta;\Gamma$ consists of a type context and a term context.

Notation 3.1 The following notational conventions will be maintained throughout this paper. x and y range over term variables, c ranges over combinators, and t ranges over terms. Γ ranges over term contexts. The usual conventions of λ-calculus concerning grouping of declared and bound variables apply (see (Barendregt, 1984)). $\mathrm{FV}(t)$ is the set of free variables of t and $e'\{e/x\}$ is the substitution of e for x in e'.

The judgment form $\Delta;\Gamma \vdash$ means that $\Delta;\Gamma$ is a well-formed context. The corresponding inference rules are

$$\text{(empty)}\ \frac{\Delta \vdash}{\Delta;\emptyset \vdash} \qquad \text{(term)}\ \frac{\Delta;\Gamma \vdash \quad \Delta \vdash \sigma}{\Delta;\Gamma, x:\sigma \vdash}\ x \notin \mathrm{DV}(\Gamma)$$

Lemma 3.2
$\Delta;\Gamma \vdash$ implies $\Delta \vdash \Gamma(x)$ for any $x \in \mathrm{DV}(\Delta)$.

Proof
By induction on the structure of its premise. $\square$

Let $\Delta_j;\Gamma_j$ be well-formed contexts for $j = 1,2$. Define $S : \Delta_1;\Gamma_1 \to \Delta_2;\Gamma_2$ to mean that $S : \Delta_1 \to \Delta_2$ is a substitution, and that $\Gamma_1(x) = \sigma$ implies

$$\Delta_2;\Gamma_2 \vdash x : S(\sigma).$$

The inference rules for assigning type schema are given in Figure 2.

$$(x)\ \frac{\Delta;\Gamma \vdash}{\Delta;\Gamma \vdash x : \sigma}\ \sigma = \Gamma(x)$$

$$(c)\ \frac{\Delta;\Gamma \vdash}{\Delta;\Gamma \vdash c : \sigma_c}$$

$$(\mathrm{app})\ \frac{\Delta;\Gamma \vdash t : \tau_1 \to \tau_2 \quad \Delta;\Gamma \vdash t_1 : \tau_1}{\Delta;\Gamma \vdash t\ t_1 : \tau_2}$$

$$(\lambda)\ \frac{\Delta;\Gamma, y : \tau_1 \vdash t\{y/x\} : \tau_2}{\Delta;\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}\ y \notin \mathrm{DV}(\Gamma)$$

$$(\mathrm{let})\ \frac{\Delta;\Gamma \vdash t_1 : \sigma_1 \quad \Delta;\Gamma, y : \sigma_1 \vdash t_2\{y/x\} : \sigma_2}{\Delta;\Gamma \vdash \mathrm{let}\ x = t_1\ \mathrm{in}\ t_2 : \sigma_2}\ y \notin \mathrm{DV}(\Gamma)$$

$$(\mathrm{App})\ \frac{\Delta;\Gamma \vdash t : \forall X : \mathrm{T}.\sigma \quad \Delta \vdash \tau : \mathrm{T}}{\Delta;\Gamma \vdash t : \sigma\{\tau/X\}}$$

$$(\Lambda)\ \frac{\Delta, Y : \mathrm{T};\Gamma \vdash t : \sigma\{Y/X\}}{\Delta;\Gamma \vdash t : \forall X : \mathrm{T}.\sigma}\ Y \notin \mathrm{DV}(\Delta)$$

$$(\mathrm{App}_n)\ \frac{\Delta;\Gamma \vdash t : \forall X : n.\sigma \quad \Delta \vdash F : n}{\Delta;\Gamma \vdash t : \sigma\{F/X\}}$$

$$(\Lambda_n)\ \frac{\Delta, Y : n;\Gamma \vdash t : \sigma\{Y/X\}}{\Delta;\Gamma \vdash t : \forall X : n.\sigma}\ Y \notin \mathrm{DV}(\Delta)$$

Fig. 2. Assigning schema

Lemma 3.3
The system defined in Figure 2 satisfies the following properties

1. If $S : \Delta_1 \to \Delta_2$ is a substitution then $\Delta_1;\Gamma \vdash$ implies $\Delta_2; S(\Gamma) \vdash$.
2. Well-typing: $\Delta;\Gamma \vdash t : \sigma$ implies $\Delta \vdash \sigma$.
3. If S is a renaming of Δ then $\Delta;\Gamma \vdash J$ implies $\Delta_S; S(\Gamma) \vdash S(J)$.

4. Type substitution: if $S : \Delta_1 \to \Delta_2$ is a substitution then $\Delta_1; \Gamma \vdash t : \sigma$ implies $\Delta_2; S(\Gamma) \vdash t : S(\sigma)$.
5. Thinning: $\Delta; \Gamma_1, \Gamma_2 \vdash J$ implies $\Delta; \Gamma_1, x : \sigma, \Gamma_2 \vdash J$ for any $x \notin \mathrm{DV}(\Gamma_1, \Gamma_2)$.
6. Term substitution: if $\Delta_1, \Delta; \Gamma_1 \vdash t : \sigma$ then $\Delta_1, \Delta_2; \Gamma_1, x : \sigma, \Gamma_2 \vdash t' : \sigma'$ implies $\Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash t'\{t/x\} : \sigma'$.

Proof
Each statement is proved by induction on the structure of its premise, in some cases using earlier statements in the lemma. □

The inference rules for assigning types (rather than schema) are given in Figure 3.

$$(x)\ \frac{\Delta; \Gamma \vdash \quad S : \Delta' \to \Delta}{\Delta; \Gamma \vdash x : S(\tau)}\ \Gamma(x) = \forall \Delta'.\tau$$

$$(c)\ \frac{\Delta; \Gamma \vdash \quad S : \Delta' \to \Delta}{\Delta; \Gamma \vdash c : S(\tau)}\ \sigma_c = \forall \Delta'.\tau$$

$$(\lambda)\ \frac{\Delta; \Gamma, y : \tau_1 \vdash t\{y/x\} : \tau_2}{\Delta; \Gamma \vdash (\lambda x.t) : \tau_1 \to \tau_2}\ y \notin \mathrm{DV}(\Gamma)$$

$$(\mathrm{app})\ \frac{\Delta; \Gamma \vdash t : \tau_1 \to \tau_2 \quad \Delta; \Gamma \vdash t_1 : \tau_1}{\Delta; \Gamma \vdash (t\ t_1) : \tau_2}$$

$$(\mathrm{let})\ \frac{\Delta, \Delta'; \Gamma \vdash t_1 : \tau_1 \quad \Delta; \Gamma, y : (\forall \Delta'.\tau_1) \vdash t_2\{y/x\} : \tau_2}{\Delta; \Gamma \vdash (\mathrm{let}\ x = t_1\ \mathrm{in}\ t_2) : \tau_2}\ y \notin \mathrm{DV}(\Gamma)$$

Fig. 3. Assigning types

When assigning types, the definition of substitutions between contexts must be modified slightly, as follows. Let $\Delta_j; \Gamma_j$ be well-formed contexts for $j = 1, 2$. Define $S : \Delta_1; \Gamma_1 \to \Delta_2; \Gamma_2$ to mean that $S : \Delta_1 \to \Delta_2$ is a substitution, and that

$$\Delta_2, \Delta; \Gamma_2 \vdash x : S(\tau)$$

whenever $\Gamma_1(x) = \forall \Delta.\tau$. Note that α-conversion is used to ensure that Δ_2 and Δ have no variables in common.

Lemma 3.4
The system of Figure 3 satisfies the following properties.

1. Well-typing: $\Delta; \Gamma \vdash t : \tau$ implies $\Delta \vdash \tau : \mathrm{T}$.
2. If $S : \Delta_1 \to \Delta_2$ is a substitution then $\Delta_1; \Gamma \vdash$ implies $\Delta_2; S(\Gamma) \vdash$.
3. Type substitution: if $S : \Delta_1 \to \Delta_2$ is a substitution then $\Delta_1; \Gamma \vdash t : \tau$ implies $\Delta_2; S(\Gamma) \vdash t : S(\tau)$.
4. If S is a renaming of Δ then $\Delta; \Gamma \vdash J$ implies $\Delta_S; S(\Gamma) \vdash S(J)$.
5. Thinning: $\Delta; \Gamma_1, \Gamma_2 \vdash J$ implies $\Delta; \Gamma_1, x : \sigma, \Gamma_2 \vdash J$ for any $x \notin \mathrm{DV}(\Gamma_1, \Gamma_2)$.

6. Term substitution: if $\Delta_1, \Delta; \Gamma_1 \vdash t : \tau$ then $\Delta_1, \Delta_2; \Gamma_1, x : (\forall\Delta.\tau), \Gamma_2 \vdash t' : \tau'$ implies $\Delta_1, \Delta_2; \Gamma_1, \Gamma_2 \vdash t'\{t/x\} : \tau'$.

Proof

Each statement is proved by induction on the structure of its premise, in some cases using earlier statements in the lemma. □

The following lemma compares inference of types with inference of schema. To distinguish them we use the notation $\Delta; \Gamma \vdash_\tau t : \tau$ for the former and $\Delta; \Gamma \vdash_\sigma t : \sigma$ for the latter.

Lemma 3.5

$\Delta; \Gamma \vdash_\tau t : \tau$ implies $\Delta; \Gamma \vdash_\sigma t : \tau$. Conversely, $\Delta; \Gamma \vdash_\sigma t : \forall\Delta'.\tau$ implies $\Delta, \Delta'; \Gamma \vdash_\tau t : \tau$ (assuming that α-conversion is used to ensure well-formedness of the latter context).

Proof

The proof is by straightforward induction on the structure of the proofs of the judgments. □

3.2 Type inference algorithm

Let Δ_1 be a type context, Γ be a term context and t be a term. A *typing* for (Δ_1, Γ, t) consists of a triple (Δ_2, S, τ) such that $S : \Delta_1 \to \Delta_2$ is a substitution and

$$\Delta_2; S(\Gamma) \vdash t : \tau.$$

A *most general typing* for (Δ_1, Γ, t) is a typing as above such that if (Δ_2', S', τ') is any other typing for it then there is a substitution $R : \Delta_2 \to \Delta_2'$ such that $R\ S = S'$ on $\mathrm{DV}(\Delta_1)$ and $R(\tau) = \tau'$.

Milner's algorithm W can be modified to produce a most general typing for our terms, whenever any typing exists. In the description of the algorithm we assume that bound variables are renamed to avoid clashes, and fresh variables are introduced whenever needed.

Theorem 3.6

Let $\Delta_1; \Gamma$ be a well-formed context.

1. Soundness: if $W(\Delta_1, \Gamma, t) = (\Delta_2, S, \tau)$ then $S : \Delta_1 \to \Delta_2$ and $\Delta_2; S(\Gamma) \vdash t : \tau$.
2. Completeness: if $S' : \Delta_1 \to \Delta_3$ and $\Delta_3; S'(\Gamma) \vdash t : \tau'$, then ($W$ succeeds and) there exists a substitution $R : \Delta_2 \to \Delta_3$ such that $S' = R\ S$ on $\mathrm{DV}(\Delta_1)$ and $\tau' = R(\tau)$.

Proof

Both statements are proved by induction on the structure of t (see (Tofte, 1988)) and use type substitution (see Lemma 3.4). □

- $W(\Delta, \Gamma, x) = (\Delta\ \Delta_1, \mathrm{id}, \tau)$, where $\Gamma(x) = \forall \Delta_1.\tau$
- $W(\Delta, \Gamma, c) = (\Delta\ \Delta_1, \mathrm{id}, \tau)$, where $\sigma_c = \forall \Delta_1.\tau$
- $W(\Delta, \Gamma, \lambda x.t) = (\Delta_1, S, SX \to \tau_2)$, where

$$(\Delta_1, S, \tau_2) = W(\Delta\ X : \mathrm{T}, \Gamma\ x : X, t)$$

- $W(\Delta, \Gamma, t\ t_1) = (\Delta_3, U\ R\ S, UX)$, where

$$\begin{array}{rcl} (\Delta_1, S, \tau) & = & W(\Delta, \Gamma, t) \\ (\Delta_2, R, \tau_1) & = & W(\Delta_1, S(\Gamma), t_1) \\ (\Delta_3, U) & = & \mathcal{U}(\Delta_2\ X : \mathrm{T}, R(\tau), \tau_1 \to X) \end{array}$$

- $W(\Delta, \Gamma, \mathrm{let}\ x = t_1\ \mathrm{in}\ t_2) = (\Delta_4, R\ S, \tau_2)$, where

$$\begin{array}{rcl} (\Delta_1, S, \tau_1) & = & W(\Delta, \Gamma, t_1) \\ \Delta_2 & = & \Delta_1 \lceil (\cup \{\mathrm{FV}(SX) | X \in \mathrm{DV}(\Delta)\}) \\ \Delta_3 & = & \Delta_1 - \Delta_2 \\ (\Delta_4, R, \tau_2) & = & W(\Delta_2, S(\Gamma)\ x : \forall \Delta_3.\tau_1, t_2) \end{array}$$

By definition Δ_2 is the smallest sub-context of Δ_1 such that $S : \Delta \to \Delta_2$, so that we will obtain $R\ S : \Delta \to \Delta_4$ as required.

Fig. 4. Algorithm W

4 Constants and Reductions

FML is a general framework for discussing functor-based polymorphism, whose detailed structure will be reflected in the choice of constants, or (combinators) of the language. These constants come in families indexed by functor arities. We will begin with the constants for mapping, as these are central to the functorial approach. To this can be added constants that correspond to introduction and elimination rules for the functor and type constructors of the language, in the style advocated in Martin-Löf type theory (Nordström *et al.*, 1990). The choice of constants and their names will be made to reflect this fact. Afterwards, we will introduce a comprehensive system of syntactic sugar that simplifies the process of constructing terms. While the fundamentals are unchanged from those of (Bellè *et al.*, 1996) the surface syntax is changed substantially.

As the FML functors can be interpreted as being shapely over lists, we can introduce more constants motivated by such an interpretation. For example, we can introduce constants that allow the extraction of a data list from a value of functor type. Also, since the FML functors all have decidable objects of shapes, we can support the zipping of values that have the same shape, e.g. lists having the same length. Further, in Section 6 we introduce case analysis over functors when defining polytypic terms, since this is validated by the inductive functor interpretation of FML functors.

It is convenient to present the reduction rules with the constants. These are used to generate a rewriting system in the usual way, by applying reductions to arbitrary sub-terms.

Beta-reduction. Of course, the basic language has the standard reduction rules of the λ-calculus with the let construct for local definitions:

$$\begin{array}{rcl} (\lambda x.t_2)\ t_1 & > & t_2\{t_1/x\} \\ \mathrm{let}\ x = t_1\ \mathrm{in}\ t_2 & > & t_2\{t_1/x\} \end{array}$$

4.1 Basic constants and FML_{basic}

In this section we fix a set of basic combinators, and call the resulting calculus FML_{basic}. The key polytypic combinator is mapping.

Mapping. There is one family of constants that express the action of functors on morphisms of a category, namely

$$\mathrm{map}_m : \forall F : m.\forall X_{i:m}, Y_{i:m} : \mathrm{T}.(X_i \to Y_i)_{i:m} \to F(\overline{X}) \to F(\overline{Y})$$

Of course, providing reduction rules for map_m is by no means trivial, as explained in the introduction. In fact it has no reduction rules of its own, but only interacts with the functor-based term constructors.

Term constructors For each functor constructor, we give the corresponding term constructors:

$$\begin{array}{rcl} \mathrm{intro}^1 & : & 1 \\ \mathrm{intro}^\times & : & \forall X_0, X_1 : \mathrm{T}.\ X_0 \to X_1 \to X_0 \times X_1 \\ \mathrm{intro}^+_j & : & \forall X_0, X_1 : \mathrm{T}.\ X_j \to X_0 + X_1 \qquad \text{for } j : 2 \\ \mathrm{intro}^\Pi_{m,i} & : & \forall X_{j:m} : \mathrm{T}.\ X_i \to \Pi^m_i(\overline{X}) \\ \mathrm{intro}^\circ_{m,n} & : & \forall F : m.\forall G_{i:m} : n.\forall X_{j:n} : \mathrm{T}.\ F(G_i(\overline{X})_{i:m}) \to F\langle \overline{G}\rangle^n(\overline{X}) \\ \mathrm{intro}^\mu_m & : & \forall F : m+1.\forall X_{i:m} : \mathrm{T}.\ F(\overline{X}, \mu F(\overline{X})) \to \mu F(\overline{X}). \end{array}$$

Mapping. Mapping preserves the structure of its data argument, i.e. it commutes with the functor-based term constructors. The reductions are:

$$\begin{array}{rcl} \mathrm{map}_0\ \mathrm{intro}^1 & > & \mathrm{intro}^1 \\ \mathrm{map}_2\ f_0\ f_1\ (\mathrm{intro}^\times\ t_0\ t_1) & > & \mathrm{intro}^\times\ (f_0\ t_0)\ (f_1\ t_1) \\ \mathrm{map}_2\ f_0\ f_1\ (\mathrm{intro}^+_j\ t) & > & \mathrm{intro}^+_j\ (f_j\ t) \\ \mathrm{map}_m\ f_{k:m}\ (\mathrm{intro}^\Pi_{m,i}\ t) & > & \mathrm{intro}^\Pi_{m,i}\ (f_i\ t) \\ \mathrm{map}_n\ f_{k:n}\ (\mathrm{intro}^\circ_{m,n}\ t) & > & \mathrm{intro}^\circ_{m,n}\ (\mathrm{map}_m\ (\mathrm{map}_n\ \overline{f})_{i:m}\ t) \\ \mathrm{map}_m\ f_{i:m}\ (\mathrm{intro}^\mu_m\ t) & > & \mathrm{intro}^\mu_m\ (\mathrm{map}_{m+1}\ \overline{f}\ (\mathrm{map}_m\ \overline{f})\ t). \end{array}$$

The rule for composite functors asserts that mapping over a composite functor, say $F\langle G\rangle^1$ is given by mapping with respect to F the result of mapping with respect to G. Also, in the rule for initial algebra functors, map_m reduces to an expression involving map_{m+1} i.e. depends on the functoriality of the original functor, as described in Section 2.

$$\begin{array}{rcl@{\qquad}rcl}
\text{apply } f\ x & = & f\ x & \text{case } t \text{ of} & = & \text{elim}^{+}\ (\lambda x.t_0)\ (\lambda y.t_1)\ t \\
\text{id} & = & \lambda x.x & \text{in}_0\ x \Rightarrow t_0 & & \\
g \circ f & = & \lambda x.g\ (f\ x) & \text{in}_1\ y \Rightarrow t_1 & & \\
\text{un} & = & \text{intro}^{1} & \text{tag}_{m,i} & = & \text{intro}^{\Pi}_{m,i} \\
\text{bang } x & = & \text{un} & \text{untag}_{m,i} & = & \text{elim}^{\Pi}_{m,i}\ \text{id} \\
\text{pair} & = & \text{intro}^{\times} & \text{pack}_{m,n} & = & \text{intro}^{\circ}_{m,n} \\
\langle t_0, t_1\rangle & = & \text{intro}^{\times}\ t_0\ t_1 & \text{unpack}_{m,n} & = & \text{elim}^{\circ}_{m,n}\ \text{id} \\
\pi_0 & = & \text{elim}^{\times}\ (\lambda x, y.x) & \text{wrap}_m & = & \text{intro}^{\mu}_{m} \\
\pi_1 & = & \text{elim}^{\times}\ (\lambda x, y.y) & \text{fold}_m & = & \text{elim}^{\mu}_{m} \\
\text{in}_j & = & \text{intro}^{+}_{j} & \text{unwrap}_m & = & \text{fold}_m\ (\text{map}_{m+1}\ \overline{\text{id}}\ \text{wrap}_m)
\end{array}$$

Fig. 5. Syntactic sugar

Term Destructors. According to type theory, given the constructors for a type, there is a canonical way to derive its destructors and computational rules:

$$\begin{array}{rcl}
\text{elim}^{1} & : & \forall X : \text{T}.\ X \to 1 \to X \\
\text{elim}^{\times} & : & \forall X_0, X_1, Y : \text{T}.\ (X_0 \to X_1 \to Y) \to X_0 \times X_1 \to Y \\
\text{elim}^{+} & : & \forall X_0, X_1, Y : \text{T}.\ (X_0 \to Y) \to (X_1 \to Y) \to X_0 + X_1 \to Y \\
\text{elim}^{\Pi}_{m,i} & : & \forall X_{j:m}, Y : \text{T}.\ (X_i \to Y) \to \Pi^{m}_{i}(\overline{X}) \to Y \\
\text{elim}^{\circ}_{m,n} & : & \forall F : m.\forall G_{i:m} : n.\forall X_{j:n}, Y : \text{T}.(F(G_i(\overline{X})_{i:m}) \to Y) \to F\langle \overline{G}\rangle^{n}(\overline{X}) \to Y \\
\text{elim}^{\mu}_{m} & : & \forall F : m+1.\forall X_{i:m}, Y : \text{T}.\ (F(\overline{X}, Y) \to Y) \to \mu F(\overline{X}) \to Y.
\end{array}$$

Each canonical destructor has a corresponding reduction rule:

$$\begin{array}{rcl}
\text{elim}^{1}\ t\ \text{intro}^{1} & > & t \\
\text{elim}^{\times}\ f\ (\text{intro}^{\times}\ t_0\ t_1) & > & f\ t_0\ t_1 \\
\text{elim}^{+}\ f_0\ f_1\ (\text{intro}^{+}_{j}\ t) & > & f_j\ t \\
\text{elim}^{\Pi}_{m,i}\ f\ (\text{intro}^{\Pi}_{m,i}\ t) & > & f\ t \\
\text{elim}^{\circ}_{m,n}\ f\ (\text{intro}^{\circ}_{m,n}\ t) & > & f\ t \\
\text{elim}^{\mu}_{m}\ f\ (\text{intro}^{\mu}_{m}\ t) & > & f\ (\text{map}_{m+1}\ (\lambda x.x)_{i:m}\ (\text{elim}^{\mu}_{m}\ f)\ t).
\end{array}$$

Most of these rules follow the familiar structure, returning a strictly simpler expression. The last of them is different, as the reduction of a fold introduces a mapping, as explained in Section 2.

4.2 Syntactic sugar

We introduces more familiar synonyms for some of the constants, and some handier syntax for some common patterns in Figure 5. The declaration $f\ x = t$ is syntactic sugar for the declaration $f = \lambda x.t$. Multiple argument parameters are handled similarly. Composition associates to the right.

4.2.1 Monads

In ML and FML a monad M can be described by given a type expression in one free variable $X : \mathrm{T} \vdash M(X) : \mathrm{T}$ and polymorphic terms of suitable type

$$\begin{array}{rcl} \mathit{val} & : & \forall X : \mathrm{T}.X \to M(X) \\ \mathit{let} & : & \forall X, Y : \mathrm{T}.(X \to M(Y)) \to M(X) \to M(Y) \end{array}$$

Note the use of slanted font to distinguish *let* from the let-construct let ... in ... of FML. In general we do not require that $M(X)$ be represented by an FML functor, since that would not allow the use of function types.

A very simple example is the Maybe monad, which is specified by

$$\begin{array}{rcl} \mathrm{Maybe}(X) & = & X + 1 \\ \mathit{val} & = & \lambda x.\, \mathrm{in}_0\ x \\ \mathit{let} & = & \lambda f, c.\, \mathrm{elim}^+\ f\ \mathrm{in}_1\ c \\ \mathit{fail} & = & \mathrm{in}_1\ \mathrm{un} \end{array}$$

where the constant *fail* represents failure.

Another example is the *parsing monad* $P_S(X)$. This is actually a family of monads parameterized with respect to a type variable S.

$$\begin{array}{rcl} P_S(X) & = & S \to (X \times S) + 1 \\ \mathit{val} & = & \lambda x, s.\, \mathrm{in}_0\ \langle x, s\rangle \\ \mathit{let} & = & \lambda f, c, s.\, \mathrm{elim}^+\ (\mathrm{elim}^\times\ (\lambda x, s.f\ x\ s))\ \mathrm{in}_1\ (c\ s) \\ \mathit{fail} & = & \lambda s.\, \mathrm{in}_1\ \mathrm{un} \end{array}$$

S should be thought of as a state, which may change during the computation. Note that $P_S(X)$, unlike Maybe, cannot be represented by an FML functor.

We introduce some syntactic sugar for the parsing monads, which will be used in Section 4.3

$$\begin{array}{rcl} [t] & = & \mathit{val}\ t \\ \mathrm{let}\, x \Leftarrow t_1\, \mathrm{in}\, t_2 & = & \mathit{let}\ (\lambda x.t_2)\ t_1 \end{array}$$

Maybe(X) and $P_1(X)$ are isomorphic, so we write inverse mediating isomorphisms $\mathrm{PARS2MAYBE} : P_1(X) \to \mathrm{Maybe}(X)$ and $\mathrm{MAYBE2PARS} : \mathrm{Maybe}(X) \to P_1(X)$.

4.2.2 Lists

Now let us consider some syntactic sugar for the list functor, $L = \mu Lu$ where $Lu = +\langle 1\langle\rangle^2, \times\rangle^2$. For any type X we have the constructors

$$\begin{array}{rcl} \mathrm{nil} & = & (\mathrm{wrap}_1 \circ \mathrm{pack}_{2,2} \circ \mathrm{in}_0 \circ \mathrm{pack}_{0,2})\ \mathrm{un} \\ \mathrm{cons}\ x\ y & = & (\mathrm{wrap}_1 \circ \mathrm{pack}_{2,2} \circ \mathrm{in}_1)\ \langle x, y\rangle \\ \mathrm{single}\ x & = & \mathrm{cons}\ x\ \mathrm{nil} \end{array}$$

$$
\begin{aligned}
\text{append } x\ y &= \text{fold}_1\ ((\text{elim}^+\ (\lambda x.\ \text{id}) \\
&\quad (\lambda y, z.\ \text{cons}\ (\pi_0\ y)\ (\pi_1\ y\ z))) \circ \text{unpack}_{2,2})\ x\ y \\
\text{flatten } x &= \text{fold}_1\ (\lambda y, z.\ \text{append}\ y\ z)\ x.
\end{aligned}
$$

The composite applied to un in defining nil is displayed as the top line of Figure 6. Let us see how the usual pattern-matching reductions for mapping and folding over lists can be recovered as composite reductions. Other inductive types are handled similarly. Let $f : X \to Y$ be a morphism. Then $\text{map}_1\ f$ nil reduces to nil by five map-reductions, as diagrammed in Figure 6, where $g = \text{map}_2\ f\ (\text{map}_1\ f)$ and $F(X) = 1\langle\rangle^2(X, LX) + X \times LX$. Similarly,

$$
\begin{tikzcd}
1() \arrow[r, "\text{pack}_{0,2}"] \arrow[d, "\text{map}_0"] & 1\langle\rangle^2(X, LX) \arrow[r, "\text{in}_0"] \arrow[d, "g"'] & F(X) \arrow[r, "\text{pack}_{2,2}"] \arrow[d, "\text{map}_2\ g\ g"'] & Lu(X, LX) \arrow[r, "\text{wrap}_1"] \arrow[d, "g"'] & LX \arrow[d, "\text{map}_1\ f"'] \\
1() \arrow[r, "\text{pack}_{0,2}"] & 1\langle\rangle^2(Y, LY) \arrow[r, "\text{in}_0"] & F(Y) \arrow[r, "\text{pack}_{2,2}"] & Lu(Y, LY) \arrow[r, "\text{wrap}_1"] & LY
\end{tikzcd}
$$

Fig. 6. $\text{map}_1 f$ nil

$$
\begin{aligned}
\text{map}_1\ f\ (\text{cons}\ h\ t) &\to (\text{wrap}_1 \circ \text{pack}_{2,2} \circ \text{in}_1)\ (\text{map}_2\ f\ (\text{map}_1\ f)\ \langle h, t\rangle) \\
&\to \text{cons}\ (f\ h)\ (\text{map}_1\ f\ t).
\end{aligned}
$$

Now consider folding over a list. Given terms $d : D$ and $g : X \times D \to D$ their (ordinary) list fold is given by

$$f = (\text{elim}^+\ (\lambda x.d)\ g) \circ \text{unpack}_{2,2} : Lu(X, D) \to D.$$

Hence, given $h : X$ and $t : LX$ we have:

$$
\begin{aligned}
\text{fold}_1\ f\ \text{nil} &\to f(\text{map}_2\ \text{id}\ (\text{fold}_1\ f)\ ((\text{pack}_{2,2} \circ \text{in}_0 \circ \text{pack}_{0,2})\ \text{un})) \\
&\to f(\text{pack}_{2,2}(\text{in}_0(\text{pack}_{0,2}\ \text{un}))) \\
&\to d \\
\text{fold}_1\ f\ (\text{cons}\ h\ t) &\to f\ (\text{map}_2\ \text{id}\ (\text{fold}_1\ f)\ ((\text{pack}_{2,2} \circ \text{in}_1\ (\text{pair}\ h\ t)))) \\
&\to f\ ((\text{pack}_{2,2} \circ \text{in}_1\ (\text{pair}\ h\ (\text{fold}_1\ f\ t)))) \\
&\to g\ (\text{pair}\ h\ (\text{fold}_1\ f\ t)).
\end{aligned}
$$

4.3 Shape-based constants and FML_{shape}

The shape-based combinators are motivated by the interpretation of FML functors as functors shapely over lists and with a decidable shape. In this case the semantics is used both to justify the addition of a polytypic combinator, and to decide when we have introduced enough combinators. We call FML_{shape} the extension of FML_{basic} with the additional combinators ltraverse and zipop.

Traversals. The FML functors are all *shapely over lists.* In particular, the ordering of the list entries determines an ordering on the data in the structure, which can be used to define traversals. A full treatment of the semantics and uses of traversals lies beyond the scope of this paper, but we introduce them now as a convenient generalization of the data extraction constant which follows.

Traversal is the process of visiting each node in a data structure, and performing some action. If we do not wish to limit the nature of this action in advance, then we could represent it using a generic computational monad (Moggi, 1991; Wadler, 1993)

$$\begin{array}{rcl} \mathrm{ltraverse}_m & : & \forall M : Monad.\forall F : m.\forall X_{i:m}, Y_{i:m} : \mathrm{T}. \\ & & \overline{(X \to MY)} \to F(\overline{X}) \to M(F(\overline{Y})). \end{array}$$

However, this would require an extension of FML with qualified kinds. Instead, we will parameterize $\mathrm{ltraverse}_m$ with respect to parsing monads $P_S(X) = S \to (X \times S) + 1$ and make use of the syntactic sugar introduced in Section 4.2.1. The constants for *left traversal* are

$$\mathrm{ltraverse}_m : \forall S : \mathrm{T}.\forall F : m.\forall X_{i:m}, Y_{i:m} : \mathrm{T}.\ \overline{(X \to P_S(Y))} \to F(\overline{X}) \to P_S(F(\overline{Y}))$$

whose evaluation rules are given by

$$\begin{array}{rcl} \mathrm{ltraverse}_0\ \mathrm{un} & > & [\mathrm{un}] \\ \mathrm{ltraverse}_2\ f_0\ f_1\ \langle t_0, t_1\rangle & > & \mathrm{let}\, y_0 \Leftarrow f_0\ t_0\, \mathrm{in} \\ & & \mathrm{let}\, y_1 \Leftarrow f_1\ t_1\, \mathrm{in}\, [\langle y_0, y_1\rangle] \\ \mathrm{ltraverse}_2\ f_0\ f_1\ (\mathrm{in}_j\ t) & > & \mathrm{let}\, y \Leftarrow f_j\ t\, \mathrm{in}\, [\mathrm{in}_j\ y] \\ \mathrm{ltraverse}_m\ f_{i:m}\ (\mathrm{tag}_{m,i}\ t) & > & \mathrm{let}\, y \Leftarrow f_i\ t\, \mathrm{in}\, [\mathrm{tag}_{m,i}\ y] \\ \mathrm{ltraverse}_n\ f_{i:n}\ (\mathrm{pack}_{m,n}\ t) & > & \mathrm{let}\, y \Leftarrow \mathrm{ltraverse}_m\ (\mathrm{ltraverse}_n\ \overline{f})_{i:m}\ t\, \mathrm{in}\, [\mathrm{pack}_{m,n}\ y] \\ \mathrm{ltraverse}_m\ f_{i:m}\ (\mathrm{wrap}_m\ t) & > & \mathrm{let}\, y \Leftarrow \mathrm{ltraverse}_{m+1}\ \overline{f}\ (\mathrm{ltraverse}_m\ \overline{f})\ t\, \mathrm{in}\, [\mathrm{wrap}_m\ y] \end{array}$$

Using the normal presentation of initial algebras (in which the recursion parameter is never to the left of an ordinary parameter in a product) left traversal corresponds to top-down traversal of the resulting tree. Then bottom-up traversal is obtained by performing a *right traversal* $\mathrm{rtraverse}_m$ which has the same type and reduction rules as ltraverse except that the rule for pairs has been changed to

$$\begin{array}{rcl} \mathrm{rtraverse}_2\ f_0\ f_1\ \langle t_0, t_1\rangle & > & \mathrm{let}\, y_1 \Leftarrow f_1\ t_1\, \mathrm{in} \\ & & \mathrm{let}\, y_0 \Leftarrow f_0\ t_0\, \mathrm{in}\, [\langle y_0, y_1\rangle] \end{array}$$

so that the recursion parameter is traversed before the others.

A specialization of ltraverse yields the operations for extracting data

$$\mathrm{extract}_{m,i} : \forall F : m.\forall X_{i:m} : \mathrm{T}.\ F(\overline{X}) \to L(X_i)$$

which is defined using the monad P_{LX_i}

$$\begin{array}{rl} \mathrm{extract}_{m,i}\ t = & \mathrm{case}\ \mathrm{ltraverse}_m \overline{f}\ t\ \mathrm{nil}\ \mathrm{of} \\ & \mathrm{in}_0\ x \Rightarrow \pi_1\ x \\ & \mathrm{in}_1\ x \Rightarrow \mathrm{nil} \end{array}$$

where $f_j : X_j \to P_{LX_i}1$ is given by $f_j\ x\ l = [\langle \mathrm{un}, \mathrm{cons}\ x\ l\rangle]$ when $j = i$, and $f_j\ x = [\mathrm{un}]$ otherwise.

Zipping. The constants

$$\begin{aligned}\mathrm{zipop}_m \quad : \quad & \forall S : \mathrm{T}.\forall F : m.\forall X_{i:m}, Y_{i:m}, Z_{i:m} : \mathrm{T}.\\ & (X_i \to Y_i \to P_S(Z_i))_{i:m} \to F(\overline{X}) \to F(\overline{Y}) \to P_S(F(\overline{Z}))\end{aligned}$$

are used to represent the zipping of two values having the same shape, and then applying the given functions to pairs of values. The reduction rules for zipop are given by

$$\begin{array}{rcl}
\mathrm{zipop}_0\ \mathrm{un}\ \mathrm{un} & > & [\mathrm{un}]\\
\mathrm{zipop}_2\ f_0\ f_1\ \langle s_0, s_1\rangle\ \langle t_0, t_1\rangle & > & \mathrm{let}\, z_0 {\Leftarrow} f_0\ s_0\ t_0\, \mathrm{in}\\
 & & \mathrm{let}\, z_1 {\Leftarrow} f_1\ s_1\ t_1\, \mathrm{in}\, [\langle z_0, z_1\rangle]\\
\mathrm{zipop}_2\ f_0\ f_1\ (\mathrm{in}_j\ s)\ (\mathrm{in}_j\ t) & > & \mathrm{let}\, z {\Leftarrow} f_j\ s\ t\, \mathrm{in}\, [\mathrm{in}_j\ z]\\
\mathrm{zipop}_2\ f_0\ f_1\ (\mathrm{in}_0\ s)\ (\mathrm{in}_1\ t) & > & \mathit{fail}\\
\mathrm{zipop}_2\ f_0\ f_1\ (\mathrm{in}_1\ s)\ (\mathrm{in}_0\ t) & > & \mathit{fail}\\
\mathrm{zipop}_m\ f_{i:m}\ (\mathrm{tag}_{m,i}\ s)\ (\mathrm{tag}_{m,i}\ t) & > & \mathrm{let}\, z {\Leftarrow} f_i\ s\ t\, \mathrm{in}\, [\mathrm{tag}_{m,i}\ z]\\
\mathrm{zipop}_n\ f_{i:n}\ (\mathrm{pack}_{m,n}\ s)\ (\mathrm{pack}_{m,n}\ t) & > & \mathrm{let}\, z {\Leftarrow} \mathrm{zipop}_m\ (\mathrm{zipop}_n\ \overline{f})_{i:m}\ s\ t\, \mathrm{in}\\
 & & [\mathrm{pack}_{m,n}\ z]\\
\mathrm{zipop}_m\ f_{i:m}\ (\mathrm{wrap}_m\ s)\ (\mathrm{wrap}_m\ t) & > & \mathrm{let}\, z {\Leftarrow} \mathrm{zipop}_{m+1}\ \overline{f}\ (\mathrm{zipop}_m\ \overline{f})\ s\ t\, \mathrm{in}\\
 & & [\mathrm{wrap}_m\ z]
\end{array}$$

Note that if we instantiate the parsing monad with 1, we obtain a type constructor isomorphic to the Maybe monad (see Section 4.2.1). Since it is sometimes useful to work directly with Maybe, we introduce the function zipop'_m:

$$\begin{aligned}\mathrm{zipop}'_m \quad : \quad & \forall F : m.\forall X_{i:m}, Y_{i:m}, Z_{i:m} : \mathrm{T}.\\ & (X_i \to Y_i \to \mathrm{Maybe}(Z_i))_{i:m} \to F(\overline{X}) \to F(\overline{Y}) \to \mathrm{Maybe}(F(\overline{Z}))\end{aligned}$$

defined as $\mathrm{zipop}'_m\ f\ s\ t = \textsc{pars2maybe}\ (\mathrm{zipop}_m\ (\lambda x, y.\textsc{maybe2pars}\ (f\ x\ y))\ s\ t)$.
From zipop′ we can define

$$\begin{aligned}\mathrm{zip}_m \quad &= \quad \mathrm{zipop}'_m\ (\mathrm{in}_0 \circ \mathrm{pair})_{i:m}\\ &: \quad \forall F : m.\forall X_{i:m} : \mathrm{T}.\forall Y_{i:m} : \mathrm{T}.F(\overline{X}) \to F(\overline{Y}) \to \mathrm{Maybe}(F(\overline{X \times Y}))\end{aligned}$$

Further, we can define a polymorphic equality. Assume that there is an equality test $\mathrm{eq}_i : X_i \to X_i \to 1 + 1$ for each X_i then define

$$\mathrm{eq}_m\ x\ y \quad = \quad \mathrm{let}\, z {\Leftarrow} (\mathrm{zipop}'_m\ \mathrm{eq}_{i:m}\ x\ y)\, \mathrm{in}\, \mathrm{in}_0\ \mathrm{un}$$

Setting all X_i to be 1 with the obvious equality, we obtain a test for shape equality $F(\overline{1}) \to F(\overline{1}) \to 1 + 1$.

Remark 4.1 We could have parameterized ltraverse and zipop with respect to any monad (with a *fail* operation) expressible in FML.

4.4 Properties of reduction

Let Fβ be the rewriting system generated by the above rules (with, or without ltraverse or zipop).

Theorem 4.2 (*SR*)
Let $t > t'$. If $\Delta;\Gamma \vdash t : \sigma$ then $\Delta;\Gamma \vdash t' : \sigma$.

Proof
Without loss of generality, one can assume that the reduction is basic and perform the proof by case analysis. In each case one has to analyze only the last rules in the derivation of the premise, using Lemma 3.4 to handle term substitutions. $\square$

Theorem 4.3 (*CR*)
Fβ on untyped terms is Church-Rosser.

Proof
Standard. The combinator reductions rules for Fβ are left-linear and non overlapping, and one can apply the result in (Aczel, 1978) (see also (Klop, 1980)). $\square$

Corollary 4.4 (*CR*)
Fβ on typable terms is Church-Rosser.

Proof
Immediate from SR and CR on untyped terms. $\square$

Theorem 4.5 (*SN*)
If $\Delta;\Gamma \vdash t : \sigma$, then t is strongly normalizing.

Proof
We prove SN for a system more powerful than FML, *functorial* F (briefly FF), which can type every term typable in FML, therefore SN for FF trivially implies SN for FML. The proof follows (Mendler, 1991) and uses semantic techniques (reducibility candidates). The details are available in (Bellè *et al.*, 1998). $\square$

5 Extended examples

In this section we present some examples to show the expressiveness of FML. FML has been presented as an extension of ML but it is rather awkward in comparison with ML because we have to deal explicitly with term constructors and destructors corresponding to functor constructors and we have to use the constant fold_m to implement primitive recursion. However we could introduce also in FML some of the nice features of ML, such as datatype definitions and pattern matching definitions, and take advantage of functors without losing the programmability. In this way we can point out when FML features, in particular functors, are really necessary.

So we give two versions of the examples. The first one uses ML-like datatype definitions, pattern matching for function definitions and FML functors only when strictly needed. The second one uses only "pure FML" and thus shows how to deal with term constructors and destructors. In each case, programs are constructed top-down, i.e. giving auxiliary functions after their use. Each function is presented by giving its type, value and description.

Syntactic sugar. In addition to the syntactic sugar defined in section 4.2, we use the following syntax:

- $M(X) = \text{Maybe}(X)$ with the usual operations on error monad: ok $x = \text{in}_0\ x$, fail $= \text{in}_1$ un.
- $Bool = M(1)$ with true = ok un, false = fail, if b then t_0 else $t_1 = \text{elim}^+(\text{elim}^1\ t_0)(\text{elim}^1\ t_1)\ b$.
- $X^2 = X \times X$.

5.1 One-side matching

Consider the constructor of a recursively defined algebra, i.e. of initial algebras for a functor. Such an algebra of closed terms can then be extended with an anonymous variable or placeholder called Any to create open terms. Every occurrence of Any in a term behaves like a fresh variable. The one-sided matching problem is to determine whether there is a substitution of terms for placeholders which creates a match between an open term and a closed term. Note that different occurrences of the placeholder in the same term can match different terms, so there is no need to consider substitutions.

Fix m, let F be a functor of arity $m+1$ and let $X_{i \in m}$ be m equality types, that is, for each $i \in m$ there exists a function $eq_i : X_i \to X_i \to Bool$.

5.1.1 One-side matching — ML-like version

Define the algebra of closed terms by

$$G = \text{Cns}\ \mathit{of}\ F(\overline{X}, G).$$

Then the algebra of open terms with placeholder Any is given by

$$G' = \text{Any} \mid \text{Cns}'\ \mathit{of}\ F(\overline{X}, G').$$

The functor F can be seen as a constructor for terms. So G is the type of terms constructed from F and G' is the type of terms constructed from F and the placeholder Any.

The function
$\mathsf{match} : G' \to G \to Bool$
match Any $(\text{Cns}\ w) = \text{true}$
$\mathsf{match}\ (\text{Cns}'\ w')\ (\text{Cns}\ w) = \mathsf{maybeBang}\ (\text{zipop}'_{m+1}\ \text{eq}_{i:m}\ \mathsf{match}\ w'\ w)$
takes a term of type G' and a term of type G. If the first term is Any then it matches any term. Otherwise the function checks if the structures of the two terms agree. Here

$$\begin{aligned}\text{zipop}'_{m+1} \quad : \quad & (X_i \to X_i \to M(1))_{i:m} \to (G' \to G \to M(1)) \to \\ & F(\overline{X}, G') \to F(\overline{X}, G) \to M(F(\overline{1}, 1))\end{aligned}$$

checks for equality of the constructors, and matching of the recursive sub-terms. Recall that $Bool = M(1)$ and that true is identified with (ok un) and false with fail. The sub-term (zipop'_{m+1} $\mathrm{eq}_{i:m}$ match w' w) fails as soon as an equality test eq_i or a recursive call to match fails. Otherwise it returns the common shape of w and w', of type $F(\overline{1}, 1)$.

The function

$\mathsf{maybeBang} : \forall X : \mathrm{T}.M(X) \to M(1)$

$\mathsf{maybeBang}\ z =\ $ case z of

$\quad \mathrm{in}_0\ x \Rightarrow \mathrm{in}_0\ \mathrm{un}$

$\quad \mathrm{in}_1\ y \Rightarrow \mathrm{fail}$

is required to get the right type after applying zipop'_{m+1}.

Note that in ML, one should write a specialization of zipop' for each particular choice of the functor F.

5.1.2 One-side matching — pure FML *version*

The data types G and G' defined above can be defined in FML respectively as $\mu F(\overline{X})$ and $\mu F'(\overline{X})$ where

$$F' = +\langle 1\langle\rangle^{m+1}, F\rangle^{m+1} : m+1.$$

The functor polymorphic function

$\mathsf{match} : \mu F'(\overline{X}) \to \mu F(\overline{X}) \to Bool$

$\mathsf{match} = \mathrm{fold}_m\ (\mathsf{f} \circ \mathsf{unft})$

is defined using induction (fold_m) on the first argument.

$\mathsf{unft} : F'(\overline{X}, Y) \to 1 + F(\overline{X}, Y)$

$\mathsf{unft}\ z =\ $ case $\mathrm{unpack}_{2,m+1}\ z$ of

$\quad \mathrm{in}_0\ x \Rightarrow \mathrm{unpack}_{0,m+1}\ x$

$\quad \mathrm{in}_1\ y \Rightarrow y$

This coerces the type $F'(\overline{X}, Y)$ to the type $1 + F(\overline{X}, Y)$.

$\mathsf{f} : 1 + F(\overline{X}, \mu F(\overline{X}) \to Bool) \to \mu F(\overline{X}) \to Bool$

$\mathsf{f}\ z\ w =\ $ case z of

$\quad \mathrm{in}_0\ x \Rightarrow \mathrm{true}$

$\quad \mathrm{in}_1\ y \Rightarrow \mathsf{maybeBang}(\mathrm{zipop}'_{m+1}\ \mathrm{eq}_{i:m}\ \mathrm{apply}\ y\ (\mathrm{unwrap}_m\ w))$

This is the core of the function match.

The function $\mathsf{maybeBang}$ is defined as in the ML-like version.

5.2 Unification

In unification literature, a term is usually defined to be either a variable or an application of a constructor to zero or more terms and possibly to some parameters. At the level of types, we identify the term constructor with a functor F of arity $m+1$, the variables with a type V and the parameters with some types $X_{i\in m}$. We suppose that all types $V, \overline{X}$ support equality, that is, there exist a function $eq_V : V \to V \to Bool$ and for each $i \in m$ there is a function $\mathrm{eq}_i : X_i \to X_i \to Bool$. We now describe a functional unification algorithm that works for all such F, V and $\overline{X}$. A list of the functions used is given in Figure 7.

5.2.1 *Unification — ML-like version*

In this section we describe the unification algorithm using ML-like datatypes for the type of terms and the type of lists:

- $LX = \mathrm{Nil} \mid \mathrm{Cons}\ of\ X * LX$
- $Term = \mathrm{Var}\ of\ V \mid \mathrm{Cns}\ of\ F(\overline{X}, Term)$

We use the following abbreviations: let $Pair = Term^2$ and let $Pairs = L(Pair)$. Moreover let $Subst = V \to Term$ be the type of substitutions, that is the functions from variables to terms. Also, id_{subst} is the identity substitution, defined as $id_{subst}\ v = \mathsf{var2term}\ v$.

The function
$\mathsf{unify} : Pairs \to M(Subst)$
$\mathsf{unify}\ \langle t_1, t_2\rangle = \mathsf{unifyl}\ (\mathrm{single}\langle t_1, t_2\rangle)\ id_{subst}$
takes two terms and gives back the most general unifier if it exists. It is implemented using:
$\mathsf{unifyl} : PairsFV, \overline{X} \to M(Subst)$
$\mathsf{unifyl}\ (\mathrm{Nil})\ s = \mathrm{ok}\ s$
$\mathsf{unifyl}\ (\mathrm{Cons}\ y\ ys)\ s = \mathrm{let}\ z \Leftarrow \mathsf{step}\langle \mathrm{Cons}\ y\ ys, s\rangle\ \mathrm{in}\ \mathsf{unifyl}\ (\pi_0\ z)(\pi_1\ z)$
which takes a list of pair of terms and updates the current substitution, given as an extra argument.

The step *function.* The core of the unification algorithm is the function step which is iterated until the list of "pairs of terms" to be unified is empty. The termination of the algorithm is guaranteed since on each pass either the number of free variables in the list of pairs decreases, or the number of constructors in the list of pairs decreases but the number of variables doesn't increase.

$\mathsf{step} : Pairs \times Subst \to M(Pairs \times Subst)$
$\mathsf{step}\ \langle \mathrm{Nil}, s\rangle = \mathrm{ok}\ \langle \mathrm{Nil}, s\rangle$
$\mathsf{step}\ \langle \mathrm{Cons}\ y\ ys, s\rangle = \mathsf{Cases}\ y\ ys\ s$

This function takes a list of pairs of terms and a substitution. If the list is empty then it does nothing. If the list has a head, then it decomposes the head in such a way it is possible to distinguish four cases depending on whether or not the terms are variables.

The function
$\mathsf{Cases} : Pair \to Pairs \to Subst \to M(Pairs \times Subst)$
$\mathsf{Cases}\ \langle \mathrm{Var}\ v_0, \mathrm{Var}\ v_1\rangle\ l\ s = \mathsf{VarVarCase}\ v_0\ v_1\ \langle l, s\rangle$
$\mathsf{Cases}\ \langle \mathrm{Var}\ v_0, \mathrm{Cns}\ w_1\rangle\ l\ s = \mathsf{VarCnsCase}\ v_0\ w_1\ \langle l, s\rangle$
$\mathsf{Cases}\ \langle \mathrm{Cns}\ w_0, \mathrm{Cns}\ v_1\rangle\ l\ s = \mathsf{VarCnsCase}\ v_1\ w_0\ \langle l, s\rangle$
$\mathsf{Cases}\ \langle \mathrm{Var}\ w_0, \mathrm{Cns}\ w_1\rangle\ l\ s = \mathsf{CnsCnsCase}\ w_0\ w_1\ \langle l, s\rangle$
determines the cases, which are handled below.

$\mathsf{VarVarCase} : V \to V \to Pairs \times Subst \to M(Pairs \times Subst)$

$$
\begin{aligned}
\mathsf{VarVarCase}\ v_0\ v_1\ \langle l, s\rangle\ = \ & \mathrm{if}\ \mathrm{eq}_V\ v_0\ v_1 \\
& \mathrm{then}\ \mathrm{ok}\ \langle l, s\rangle \\
& \mathrm{else}\ \mathrm{ok}\ (\mathsf{subUpdate}\ v_0\ (\mathrm{Var}\ v_1)\ l\ s)
\end{aligned}
$$

The function VarVarCase deals with the case in which both components are variables. If they are equal it does nothing, otherwise it augments the substitution and applies the new substitution to all the terms in the list.

$$\mathsf{VarCnsCase} : V \to F(\overline{X}, \mathit{Term}) \to \mathit{Pairs} \times \mathit{Subst} \to M(\mathit{Pairs} \times \mathit{Subst})$$
$$\begin{aligned}\mathsf{VarCnsCase}\ v\ w\ \langle l, s\rangle \ =\ & \text{if } \mathsf{occCheck}\ v\ (\mathrm{Cns}\ w) \\ & \text{then fail} \\ & \text{else ok } (\mathsf{subUpdate}\ v\ (\mathrm{Cns}\ w)\ l\ s)\end{aligned}$$

If either component is a variable then VarCnsCase does the usual occurs check, then it augments the substitution and applies the new substitution to all the terms in the list.

$$\mathsf{CnsCnsCase} : F(\overline{X}, \mathit{Term}) \to F(\overline{X}, \mathit{Term}) \to \mathit{Pairs} \times \mathit{Subst} \to M(\mathit{Pairs} \times \mathit{Subst})$$
$$\mathsf{CnsCnsCase}\ w_0\ w_1\ \langle l, s\rangle \ =$$
$$\text{let } z \Leftarrow (\mathrm{zipop}'_{m+1}\ \mathrm{eq}_{i:m}\ \mathrm{pair}\ w_0\ w_1) \text{ in ok } \langle \mathrm{append}\ (\mathrm{extract}_{m+1,m}\ z)\ l\ , s\rangle$$

Otherwise, by using zipop'_{m+1}, CnsCnsCase compares the top level of the two terms and if it succeeds, it extracts the subterm pairs and updates the list.

Auxiliary functions. In this paragraph we define the auxiliary functions used in the implementation of step.

$$\mathsf{occCheck} : V \to \mathit{Term} \to \mathit{Bool}$$
$$\mathsf{occCheck}\ v\ t = \mathsf{occCLml}\ v\ (\mathrm{single}\ t)$$

This is implemented in terms of occCLml that takes a list of terms instead of a term.

$$\mathsf{occCLml} : V \to L(\mathit{Term}) \to \mathit{Bool}$$
$$\mathsf{occCLml}\ v\ \mathrm{Nil} = \mathrm{false}$$
$$\mathsf{occCLml}\ v\ (\mathrm{Cons}\ (\mathrm{Var}\ v_0)\ ys) = (\mathrm{eq}_V\ v\ v_0) \text{ or } (\mathsf{occCLml}\ v\ ys)$$
$$\mathsf{occCLml}\ v\ (\mathrm{Cons}\ (\mathrm{Cns}\ w)\ ys) = \mathsf{occCLml}\ v\ (\mathrm{append}\ (\mathrm{extract}_{m+1,m}\ w)\ ys)$$

This function checks if a variable occurs in a list of terms. If the list of terms is not empty, it takes the term in the head of the list and if this term is a variable compare the two variables and continue the check on the rest of the list, otherwise the list of variables occurring in that term is extracted and added to the list of terms.

$$\mathsf{subUpdate} : V \to \mathit{Term} \to \mathit{Pairs} \to \mathit{Subst} \to \mathit{Pairs} \times \mathit{Subst}$$
$$\mathsf{subUpdate}\ v\ t\ l\ s\ = \langle \mathsf{subInst}\ v\ t\ l, \mathsf{subComp}\ v\ t\ s\rangle$$

This function pairs subInst and subComp.

$$\mathsf{subComp} : V \to \mathit{Term} \to \mathit{Subst} \to \mathit{Subst}$$
$$\mathsf{subComp}\ v\ t\ s\ w = \text{if } \mathrm{eq}_V\ w\ v \text{ then } t \text{ else } \mathsf{subst}\ v\ t\ (s\ w)$$

This function updates a substitution, once it is given a pair variable-term (elementary substitution).

$$\mathsf{subInst} : V \to \mathit{Term} \to \mathit{Pairs} \to \mathit{Pairs}$$
$$\mathsf{subInst}\ v\ t\ \mathrm{Nil} = \mathrm{Nil}$$
$$\mathsf{subInst}\ v\ t\ (\mathrm{Cons}\ y\ ys) = \mathrm{Cons}\ \langle \mathsf{subst}\ v\ t\ (\pi_0\ y), \mathsf{subst}\ v\ t\ (\pi_1\ y)\rangle\ (\mathsf{subInst}\ v\ t\ ys)$$

This function applies an elementary substitution to a list of pairs of terms.

$$\mathsf{subst} : V \to \mathit{Term} \to \mathit{Term} \to \mathit{Term}$$

$\mathsf{subst}\ v\ t\ (\mathrm{Var}\ v) = \mathrm{if}\ \mathrm{eq}_V\ x\ v\ \mathrm{then}\ t\ \mathrm{else}\ \mathrm{Var}\ v$
$\mathsf{subst}\ v\ t\ (\mathrm{Cns}\ w) = \mathrm{Cns}\ (\mathrm{map}_{m+1}\mathrm{id}_{i:m}\ (\mathsf{subst}\ v\ t\ w))$
This applies an elementary substitution to a term.

5.2.2 Unification — pure FML *version*

To define the FML-type corresponding to the ML-like datatypes defined in the previous section we need to define the following functors and types:

- $Tu = +\langle \Pi_0^{m+2}, F\langle \Pi_1^{m+2}, \ldots, \Pi_{m+1}^{m+2}\rangle^{m+2}\rangle^{m+2} : m+2$ is the functor that maps types $V, \overline{X}$ and Y to $V + F(\overline{X}, Y)$.
- $T = \mu Tu : m+1$.
- $Term = T(V, \overline{X})$ is the type of terms with variables in V and term constructors given by the functor F.
- $Subst = V \to Term$ is the type of substitutions, that is the functions from variables to terms.
- $Pair = Term^2$.
- $Pairs = L(Pair)$.

For lists we use the list functor, L in section 4.2.2. Recall that $L = \mu Lu$ where $Lu = +\langle 1\langle\rangle^2, \times\rangle^2$ and that the usual operations on lists are nil, cons, single, append, flatten.

The function unify is defined as in the previous ML-like version:

$\mathsf{unify} : Pair \to M(Subst)$
$\mathsf{unify}\ \langle t_1, t_2\rangle = \mathsf{unifyl}\ (\mathrm{single}\langle t_1, t_2\rangle)\ id_{subst}$

However, the function unifyl now needs to be redefined, since the ML-like implementation uses a general recursive constructor whereas in FML we have only primitive recursion.

$\mathsf{unifyl} : Pairs \to Subst \to M(Subst)$
$\mathsf{unifyl}\ l\ s\ = \mathrm{let}\, p \Leftarrow \mathsf{outIter}\ (\mathsf{varNumL}\ l)\ \langle l, s\rangle\, \mathrm{in}\, \mathrm{ok}\ (\pi_1\ p)$

The implementation now requires two loops to be iterated. The outermost is bounded by the number of variables occurring in the list of pairs (for simplicity we take the number of occurrences), the innermost by the number of constructors occurring in the list of pairs. In fact, in order to guarantee that on each outermost iteration the number of variables decreases, we have only to iterate the basic step at most a number of times equal to the number of term constructors in the list. This occurs when the two terms have the same structure.

The function

$\mathsf{outIter} : L(1) \to Pairs \times Subst \to M(Pairs \times Subst)$

$$
\begin{aligned}
\mathsf{outIter} = \mathrm{fold}_1(\lambda z.\ &\mathrm{case}\ \mathsf{flatunflist}\ z\ \mathrm{of}\\
&\mathrm{in}_0\ x \Rightarrow \mathrm{ok}\\
&\mathrm{in}_1\ y \Rightarrow \mathsf{innIter}(\mathsf{cnsNumL}\ l))
\end{aligned}
$$

implements the outer loop, whereas the function

$\mathsf{innIter} : L(1) \to Pairs \times Subst \to M(Pairs \times Subst)$

$$\begin{aligned}\mathsf{innIter} = \mathrm{fold}_1(\lambda z.\ & \text{case } \mathsf{flatunflist}\ z \text{ of} \\ & \mathrm{in}_0\ x \Rightarrow \mathrm{ok} \\ & \mathrm{in}_1\ y \Rightarrow \mathsf{step})\end{aligned}$$

implements the inner loop.

Counting variables and constructors. The natural number type is implemented as $L(1)$ so iteration can be mimicked by folding on a list.

$\mathsf{varNumL} : \mathit{Pairs} \to L(1)$
$\mathsf{varNumL} = (\mathrm{map}_1\ \mathrm{bang}) \circ \mathrm{extract}_{m+1,0} \circ \mathrm{pack}_{2,m+1} \circ \mathrm{pack}_{1,2}$

The number of variable occurrences in the list of pairs is computed by the function $\mathrm{extract}_{m+1,0}$ (defined in section 4.3), which extracts the data of type V from the data structure $L\langle\times\langle T,T\rangle\rangle(V,\overline{X})$. Note that $(\mathrm{pack}_{2,m+1} \circ \mathrm{pack}_{1,2})$ coerces $L(\mathit{Term}^2)$ to $L\langle\times\langle T,T\rangle\rangle(V,\overline{X})$.

$\mathsf{cnsNumL} : \mathit{Pairs} \to L(1)$

$$\begin{aligned}\mathsf{cnsNumL} = \mathrm{fold}_1(\lambda z.\ & \text{case } \mathsf{flatunflist}\ z \text{ of} \\ & \mathrm{in}_0\ x \Rightarrow \mathrm{nil} \\ & \mathrm{in}_1\ \langle y, ys\rangle \Rightarrow \\ & \mathrm{append}\ (\mathrm{append}\ (\mathsf{cnsNum}(\pi_0 y))\ (\mathsf{cnsNum}(\pi_1 y)))\ ys)\end{aligned}$$

The number of constructors in the list of pairs is obtained by adding the number of constructors occurring in each term. This, in turn, is computed recursively adding the number of constructors of the subterms.

$\mathsf{cnsNum} : \forall G : m.\mu G(\overline{X}) \to L(1)$
$\mathsf{cnsNum} = \mathrm{fold}_{m+1}(\lambda z.\mathrm{append}\ (\mathrm{single}\ un)(\mathrm{flatten}\ (\mathrm{extract}_{m+1,m}\ z)))$

This function uses the $\mathrm{extract}_{m+1,m}$ function to locate the constructors inside a term.

The step *function.* The core of the algorithm is given by the function step which reduces to cases, as before.

$\mathsf{step} : \mathit{Pairs} \times \mathit{Subst} \to M(\mathit{Pairs} \times \mathit{Subst})$

$$\begin{aligned}\mathsf{step}\ \langle l, s\rangle = \ & \text{case } \mathsf{unfoldlist}\ l \text{ of} \\ & \mathrm{in}_0\ x \Rightarrow \mathrm{ok}\ \langle l, s\rangle \\ & \mathrm{in}_1\ \langle y, ys\rangle \Rightarrow \ \text{case } \mathsf{unfoldterm}(\pi_0 y) \text{ of} \\ & \qquad \mathrm{in}_0\ v_0 \Rightarrow \ \text{case } \mathsf{unfoldterm}(\pi_1 y) \text{ of} \\ & \qquad\qquad \mathrm{in}_0\ v_1 \Rightarrow \mathsf{VarVarCase}\ v_0\ v_1\ \langle ys, s\rangle \\ & \qquad\qquad \mathrm{in}_1\ w_1 \Rightarrow \mathsf{VarCnsCase}\ v_0\ w_1\ \langle ys, s\rangle \\ & \qquad \mathrm{in}_1\ w_0 \Rightarrow \ \text{case } \mathsf{unfoldterm}(\pi_1 y) \text{ of} \\ & \qquad\qquad \mathrm{in}_0\ v_1 \Rightarrow \mathsf{VarCnsCase}\ v_1\ w_0\ \langle ys, s\rangle \\ & \qquad\qquad \mathrm{in}_1\ w_1 \Rightarrow \mathsf{CnsCnsCase}\ w_0\ w_1\ \langle ys, s\rangle\end{aligned}$$

The following functions deal with the three cases:

$\mathsf{VarVarCase} : V \to V \to \mathit{Pairs} \times \mathit{Subst} \to M(\mathit{Pairs} \times \mathit{Subst})$

$$\begin{aligned}\mathsf{VarVarCase}\ v_0\ v_1\ \langle l, s\rangle \ = \ & \text{if } \mathrm{eq}_V\ v_0\ v_1 \\ & \text{then ok } \langle l, s\rangle \\ & \text{else ok } (\mathsf{subUpdate}\ v_0\ (\mathsf{var2term}\ v_1)\ l\ s)\end{aligned}$$

$\mathsf{VarCnsCase} : V \to F(\overline{X}, \mathit{Term}) \to \mathit{Pairs} \times \mathit{Subst} \to M(\mathit{Pairs} \times \mathit{Subst})$

$$\begin{array}{rcl}\mathsf{VarCnsCase}\ v\ w\ \langle l,s\rangle & = & \text{if } \mathsf{occCheck}\ v\ (\mathsf{cns2term}\ w)\\ & & \text{then fail}\\ & & \text{else ok } (\mathsf{subUpdate}\ v\ (\mathsf{cns2term}\ w)\ l\ s)\end{array}$$

$\mathsf{CnsCnsCase} : F(\overline{X}, \mathit{Term}) \to F(\overline{X}, \mathit{Term}) \to \mathit{Pairs} \times \mathit{Subst} \to M(\mathit{Pairs} \times \mathit{Subst})$
$\mathsf{CnsCnsCase}\ w_0\ w_1\ \langle l,s\rangle\ =$
$\text{let } z \Leftarrow (\text{zipop}_{m+1}\ \text{eq}_{i:m}\ \text{pair}\ w_0\ w_1) \text{ in ok } \langle \text{append } (\text{extract}_{m+1,m}\ z)\ l\ , s\rangle$

Auxiliary functions. In this paragraph we define the auxiliary functions used in the implementation of step.

$\mathsf{occCheck} : V \to \mathit{Term} \to \mathit{Bool}$
$\mathsf{occCheck}\ v\ t = \mathsf{occCheckL}\ v\ (\text{extract}_{m+1,0}\ t)$

This function checks if a variable occurs in a term, by checking if the variable occurs in the list of the variables extracted from the term.

$\mathsf{occCheckL} : X \to L(X) \to \mathit{Bool}$

$$\begin{array}{l}\mathsf{occCheckL}\ a = \text{fold}_1(\lambda z.\ \text{case } \mathsf{flatunflist}\ z \text{ of}\\ \qquad\qquad \text{in}_0\ x \Rightarrow \text{false}\\ \qquad\qquad \text{in}_1\ \langle y, ys\rangle \Rightarrow\ \text{if eq}_X\ y\ x\\ \qquad\qquad\qquad \text{then true}\\ \qquad\qquad\qquad \text{else } y)\end{array}$$

This function checks if a value occurs in a list.

$\mathsf{subUpdate} : V \to \mathit{Term} \to \mathit{Pairs} \to \mathit{Subst} \to \mathit{Pairs} \times \mathit{Subst}$
$\mathsf{subUpdate}\ v\ t\ l\ s\ = \langle \mathsf{subInst}\ v\ t\ l, \mathsf{subComp}\ v\ t\ s\rangle$

This function pairs the functions $\mathsf{subInst}$ and $\mathsf{subComp}$.

$\mathsf{subComp} : V \to \mathit{Term} \to \mathit{Subst} \to \mathit{Subst}$

$$\begin{array}{l}\mathsf{subComp}\ v\ t\ s\ = \lambda w.\ \text{if eq}_V\ w\ v\\ \qquad\qquad \text{then } t\\ \qquad\qquad \text{else } \mathsf{subst}\ v\ t\ (s\ w)\end{array}$$

This function updates a substitution, once it is given a pair variable-term (elementary substitution).

$\mathsf{subInst} : V \to \mathit{Term} \to \mathit{Pairs} \to \mathit{Pairs}$

$$\begin{array}{l}\mathsf{subInst}\ v\ t = \text{fold}_1(\lambda z.\ \text{case } \mathsf{flatunflist}\ z \text{ of}\\ \qquad\qquad \text{in}_0\ x \Rightarrow \text{nil}\\ \qquad\qquad \text{in}_1\ \langle y, ys\rangle \Rightarrow \mathit{cons}\ \langle \mathsf{subst}\ v\ t\ (\pi_0\ y), \mathsf{subst}\ v\ t\ (\pi_1\ y)\rangle\ ys)\end{array}$$

This function applies an elementary substitution to a list of pairs of terms.

$\mathsf{subst} : V \to \mathit{Term} \to \mathit{Term} \to \mathit{Term}$

$$\begin{array}{l}\mathsf{subst}\ v\ t = \text{fold}_m(\lambda z.\ \text{case } \mathsf{flatunfterm}\ z \text{ of}\\ \qquad\qquad \text{in}_0\ x \Rightarrow\ \text{if eq}_V\ x\ v\\ \qquad\qquad\qquad \text{then } t\\ \qquad\qquad\qquad \text{else } \mathsf{var2term}\ x\\ \qquad\qquad \text{in}_1\ y \Rightarrow \mathsf{cns2term}\ y)\end{array}$$

This function applies an elementary substitution to a term.

Coercion functions. The following functions operate a coercion between types. They make large use of the canonical functor constructors and destructors.

$\mathsf{var2term} : V \to \mathit{Term}$
$\mathsf{var2term} = \mathrm{wrap}_{m+1} \circ \mathrm{pack}_{2,m+2} \circ \mathrm{in}_0 \circ \mathrm{tag}_{m+1,0}$
This function coerces a variable to a term.

$\mathsf{cns2term} : F(\overline{X}, \mathit{Term}) \to \mathit{Term}$
$\mathsf{cns2term} = \mathrm{wrap}_{m+1} \circ \mathrm{pack}_{2,m+2} \circ \mathrm{in}_1 \circ \mathrm{pack}_{m+1,m+2} \circ (\mathrm{map}_{m+1}(\mathrm{tag}_{m+1,i})_{i:[1..m+1]})$
This function coerces an applied constructor to a term.

$\mathsf{flatunfterm} : \mathit{Tu}(V, \overline{X}, Y) \to V + F(\overline{X}, Y)$

$$\begin{array}{ll} \mathsf{flatunfterm}\ z = & \mathrm{case}\ (\mathrm{unpack}_{2,m+2}\ z)\ \mathrm{of} \\ & \mathrm{in}_0\ x \Rightarrow \mathrm{in}_0\ (\mathrm{untag}_{m+2,0}\ x) \\ & \mathrm{in}_1\ y \Rightarrow \mathrm{unpack}_{m+1,m+2}\ (\mathrm{map}_{m+1}(\mathrm{untag}_{m+2,i})_{i:[1..m+1]}\ y) \end{array}$$

This function coerces the type $\mathit{Tu}(V, \overline{X}, Y)$ to the type $V + F(\overline{X}, Y)$.

$\mathsf{unfoldterm} : \mathit{Term} \to V + F(\overline{X}, \mathit{Term})$
$\mathsf{unfoldterm} = \mathsf{flatunfterm} \circ \mathrm{unwrap}_{m+1}$
This function unfolds a term and coerces it to the type $V + F(\overline{X}, \mathit{Term})$.

$\mathsf{flatunflist} : \mathit{Lu}(X, Y) \to 1 + X \times Y$

$$\begin{array}{ll} \mathsf{flatunflist}\ z = & \mathrm{case}\ (\mathrm{unpack}_{2,2}\ z)\ \mathrm{of} \\ & \mathrm{in}_0\ x \Rightarrow \mathrm{in}_0\ (\mathrm{unpack}_{0,2}\ x) \\ & \mathrm{in}_1\ y \Rightarrow \mathrm{in}_1\ y \end{array}$$

This function coerces the type $\mathit{Lu}(X, Y)$ to the type $1 + X \times Y$.

$\mathsf{unfoldlist} : L(X) \to 1 + X \times L(X)$
$\mathsf{unfoldlist} = \mathsf{flatunflist} \circ \mathrm{unwrap}_1$
This function unfolds a list and coerces it to the type $1 + X \times L(X)$.

6 Variations and extensions to FML

FML provides a general framework for studying a new form of polymorphism centered around the notion of *functor*. In fact, the key new feature of FML is the introduction of functor expressions, rather than the new form of polymorphism. In designing FML we have intentionally kept the interaction between functors and types to a minimum, in particular the language of functor expressions is given independently from that of types and terms. As shown in Section 2.5, functor expressions can be interpreted in many possible ways, and none of them is a priori preferable than others. In this section we propose possible variations and extensions to FML. They can be classified into three groups, according to the source of their motivation.

- type theory: for instance, choosing between terms à la Curry or à la Church, identifying types and type schema (as in system F), or introducing higher order kinds (as in system $F\omega$).
- semantics: for instance, FML_{shape} (see Section 4.3).
- programming languages: for instance, various forms of polymorphism, recursive definitions, or polytypic definitions.

First we give a brief overview of generic, type-theoretic variations and extensions, and then we discuss in greater details two programming language extensions: re-

In both versions:

- unify : $Pair \to M(Subst)$
- unifyl : $Pairs \to Subst \to M(Subst)$
- step : $Pairs \times Subst \to M(Pairs \times Subst)$
- VarVarCase : $V \to V \to Pairs \times Subst \to M(Pairs \times Subst)$
- VarCnsCase : $V \to F(\overline{X}, Term) \to Pairs \times Subst \to M(Pairs \times Subst)$
- CnsCnsCase : $F(\overline{X}, Term) \to F(\overline{X}, Term) \to Pairs \times Subst \to M(Pairs \times Subst)$
- occCheck : $V \to Term \to Bool$
- occCheckL : $X \to L(X) \to Bool$
- subUpdate : $V \to Term \to Pairs \to Subst \to Pairs \times Subst$
- subComp : $V \to Term \to Subst \to Subst$
- subInst : $V \to Term \to Pairs \to Pairs$
- subst : $V \to Term \to Term \to Term$

In ML-like version:

- Cases : $Pair \to Pairs \to Subst \to M(Pairs \times Subst)$
- occCLml : $V \to L(Term) \to Bool$

In pure FML version:

- outIter : $L(1) \to Pairs \times Subst \to M(Pairs \times Subst)$
- innIter : $L(1) \to Pairs \times Subst \to M(Pairs \times Subst)$
- varNumL : $Pairs \to L(1)$
- cnsNumL : $Pairs \to L(1)$
- cnsNum : $\forall G : m + 1.\mu G(\overline{X}) \to L(1)$
- var2term : $V \to Term$
- cns2term : $F(\overline{X}, Term) \to Term$
- flatunfterm : $Tu(V, \overline{X}, Y) \to V + F(\overline{X}, Y)$
- unfoldterm : $Term \to V + F(\overline{X}, Term)$
- flatunflist : $Lu(X, Y) \to 1 + X \times Y$
- unfoldlist : $L(X) \to 1 + X \times L(X)$

Where:

- $Subst = V \to Term$
- $Pair = Term^2$
- $Pairs = L(Pair)$

Fig. 7. Types of the functions used in the unification algorithm

cursive definitions of polymorphic expressions, and polytypic definitions. These extensions will be embodied in FML$_{poly}$ in Section 6.3.

Both extensions exploit in an essential way shape polymorphism, i.e. quantification over functors, and moreover they require another form of polymorphism, namely quantification over functor arities $m : N$. A formal account of quantification over functor arities involves some technical subtleties with dependent types (well-known to people working on logical frameworks based on type theory), which are beyond the scope of this paper. Instead we give an informal presentation, in which the finitary nature of syntax and inference rules is lost.

6.1 Type-theoretic variations and extensions

We consider some simple variations on terms which make sense for other typed calculi. They are: terms à la Church; some extensions suggested by system F and $F\omega$ as extensions of ML; and an extension of FML with the inductive kind of natural numbers.

6.1.1 Terms à la Church

The syntax of terms à la Curry is parameterized with respect to constants c

$$t \quad ::= \quad x \mid c \mid \lambda x.t \mid t_1\ t_2 \mid \text{let } x = t_1 \text{ in } t_2$$

A more general approach is to parameterize terms with respect to binders c

$$\begin{array}{lcll} t & ::= & x \mid c(\overline{f}) & \text{terms} \\ f & ::= & [\overline{x}].t & \text{abstractions} \end{array}$$

Using abstraction and application one could replace binders with higher-order constants, but binders are preferable in the absence of functional types. For instance, application of the constant map_m for mapping applied to some $\lambda x.t_i$ and t is replaced by the binder $\text{map}_m(([x].t_i)_{i:m}, t)$.

Another variation is to use terms à la Church, i.e. with explicit type information

$$t \quad ::= \quad x \mid c \mid \lambda x : \tau.t \mid t_1\ t_2 \mid \Lambda X : \mathrm{T}.t \mid t\ \tau \mid \Lambda X : m.t \mid t\ F \mid \text{let } x : \sigma = t_1 \text{ in } t_2$$

Terms à la Church are rather cumbersome to write, but the extra information can be very useful, as advocated by (Harper & Morrisett, 1995; Aditya *et al.*, 1994; Tolmach, 1994). For instance, terms à la Church are essential to describe the most general pattern for polytypic definitions, where functor information is used (as in ad hoc polymorphism) to choose among incompatible alternatives. Terms à la Church are also useful to ensure decidability of type-checking in some extensions of FML.

6.1.2 F-like extensions

By analogy with system F we could identify types and type schema i.e.

$$\tau \quad ::= \quad X \mid F(\overline{\tau}) \mid \tau_1 \to \tau_2 \mid \forall X : \mathrm{T}.\tau \mid \forall X : m.\tau$$

(One could also contemplate identifying types with functors of arity 0 but this would make the languages of functor expressions dependent on that of type expressions.) A more substantial extension, which could be called $FF\omega$, is suggested by system $F\omega$. In this extension one can make explicit the level of kinds k, which include T and functor arities m, and add higher order kinds

$$\begin{array}{lcl} k & ::= & \mathrm{T} \mid m \mid k_1 \to k_2 \\ u & ::= & X \mid C \mid \lambda X : k.u \mid u_1\ u_2 \\ t & ::= & x \mid c \mid \lambda x : \tau.t \mid t_1\ t_2 \mid \Lambda X : k.t \mid t\ u \end{array}$$

Here u represents constructors, and t terms. In $FF\omega$ functor constructors and the type constructor $\rightarrow$ can be viewed as constants C of higher order kind, and functors of kind m coexist with m-ary type constructors of kind $\mathrm{T}^m \rightarrow \mathrm{T}$.

6.1.3 Inductive kinds

A more novel extension is to add polymorphism over functor arities. In FML it would support a single combinator for mapping

$$\mathrm{map} \quad : \quad \forall m : N.\forall F : m.\forall X_{i:m}, Y_{i:m} : \mathrm{T}.(X_i \rightarrow Y_i)_{i:m} \rightarrow F(\overline{X}) \rightarrow F(\overline{Y})$$

This requires quantification over arities in type schema

$$\sigma \quad ::= \quad \ldots \mid \forall m : N.\sigma_m$$

with the corresponding terms and reductions. A formal presentation of arity quantification would involve rules such as

$$(\forall_N)\ \frac{\Delta, m : N \vdash \sigma}{\Delta \vdash \forall m : N.\sigma}\ .$$

However, one would have to spell out what are the expressions of kind N, and how they could be used in other syntactic categories. This could be done in the setting of Martin-Löf type theory, by taking N as the inductive kind of natural numbers, which could then be used to define kinds, constructors and terms by induction. For instance, in system $F\omega$ we could define the family of kinds $\mathrm{T}^m \rightarrow \mathrm{T}$ by induction on $m : N$.

Here we give an informal description based on the following rule:

$$(\forall_N)\ \frac{\Delta \vdash \sigma_m \quad m : N}{\Delta \vdash \forall m : N.\sigma_m}\ .$$

This rule says that given an infinite family of type schema, we can form a type schema (of infinite size!). Of course, we can arbitrarily restrict the arities to any particular limit, e.g. 3 to recover finitariness and formality.

The syntax of terms and the typing rules are extended as follows

$$t \quad ::= \quad \ldots \mid (\Lambda m.t_m) \mid t_e$$

where m is a numeric variable, and e is a numeric expression.

$$(\mathrm{App}_N)\ \frac{\Delta; \Gamma \vdash t : \forall m : N.\sigma_m}{\Delta; \Gamma \vdash t_e : \sigma_e}$$

$$(\Lambda_N)\ \frac{\Delta; \Gamma \vdash t_m : \sigma_m \quad m : N}{\Delta; \Gamma \vdash (\Lambda m.t_m) : \forall m : N.\sigma_m}$$

The reduction rule for these term-constructors is

$$(\Lambda m.t_m)_e \quad > \quad t_e$$

6.1.4 Recursion

There are two ways of adding recursive definitions to ML. The first option, taken in (lazy) ML, is to introduce a fix-point combinator

$$\mathrm{fix} : \forall X : \mathrm{T}.(X \to X) \to X$$

with the usual reduction

$$\mathrm{fix}\ t > t\ (\mathrm{fix}\ t).$$

The addition of such combinator does not affect the type-inference algorithm.

The second option, taken in ML$^+$ (see (Kfoury & Tiuryn, 1992)), allows polymorphic recursion with the usual unfolding as reduction:

$$(\mathrm{rec})\ \frac{\Delta; \Gamma, x : \sigma \vdash t : \sigma}{\Delta; \Gamma \vdash (\mu x.t) : \sigma}$$

$$(\mu x.t) > t\{(\mu x.t)/x\}.$$

This second option is strictly more powerful (i.e. more terms are typable), but typability becomes undecidable. Of course, when types and type schema are identified (as in system F), the two extensions are equivalent.

In FML we adopt the second option. To achieve decidable type checking in a (formal presentation of) this extension, it is better to adopt a formulation of FML à la Church, i.e. with explicit type and functor information in terms.

6.2 Polytypic definitions

The combinators for mapping, folding, traversing, zipping, etc. capture fundamental properties common to large classes of functors. It is obviously desirable that programmers be able to construct these and other examples directly, using recursive definitions. In ML + fix one can define primitive recursion on inductive datatypes in terms of pattern matching and general recursion. By analogy, one might expect that in FML + fix one could define fold_m and map_m in terms of pattern matching for inductive types given by

$$\mathrm{match}^{\mu}_m \quad : \quad \forall F : m+1.\forall X_{i:m}, Y : \mathrm{T}.(F(\overline{X}, \mu F(\overline{X})) \to Y) \to \mu F(\overline{X}) \to Y$$

$$\mathrm{match}^{\mu}_m\ f\ (\mathrm{intro}^{\mu}_m\ t) \quad > \quad f\ t$$

Indeed, one can define fold_m and map_m for any closed functor expression F (by induction on the structure of F), but there is no way to define them *uniformly* for every functor F. There are two reasons why fix does not suffices for defining a combinator like map_m. First, we need mutually recursive definitions of families of combinators $c_{m:N}$ indexed by the natural numbers, since map_m is defined in terms of map_n. Second, we need induction on the structure of inductive functors, i.e. polytypic definitions (Sheard, 1993; Hook & Sheard, 1993; Jeuring, 1995; Meertens, 1996; Jansson & Jeuring, 1997).

For the first, we need inductive kinds, so that the type of a polytypic program can be presented as $\forall m : N.\forall F : m.\sigma_m(F)$. For the second, we require a new term formation rule poly $[-]$ to perform case analysis on functors. Finally, to unleash the power of this rule we require the stronger, polymorphic recursion.

The polytypic construction must cover the following four possibilities:

- constant functors C
- projection functors Π_i^m
- compositions $F\langle\overline{G}\rangle^m$ of functors
- initial algebra functors μF.

Here is the corresponding rule.

$$(\text{poly})\ \frac{\begin{array}{lr} \Delta;\Gamma \vdash t_C : \sigma_{n_C}(C) & C \\ \Delta;\Gamma \vdash t^{\Pi}_{m,i} : \sigma_m(\Pi_i^m) & i : m : N \\ \Delta, F : n, G_{i:n} : m; \Gamma \vdash t^{\circ}_{n,m} : \sigma_m(F\langle\overline{G}\rangle^m) & m,n : N \\ \Delta, F : m+1; \Gamma \vdash t^{\mu}_m : \sigma_m(\mu F) & m : N \end{array}}{\Delta;\Gamma \vdash \text{poly}\ \left[t_C, t^{\Pi}_{m,i}, t^{\circ}_{n,m}, t^{\mu}_m\right] : \forall m : N.\forall F : m.\sigma_m(F)}$$

Note that when applying this rule there must be a separate constant t_C for each constant functor C, and that the other premises represent infinite families.

In the formulation of FML à la Curry, i.e. without explicit type and functor information in terms, it is impossible to have reduction rules for poly $\left[t_C, t^{\Pi}_{m,i}, t^{\circ}_{n,m}, t^{\mu}_m\right]$, which satisfy SR (Subject Reduction) and other syntactic properties. For example, it is easy to construct a combinator $is_C : \forall m : N.\forall F : n.Bool$, which tests whether F is the constant functor C. However, the most natural reductions ($is_C >$ true and $is_C >$ false) violate CR (Church-Rosser).

Therefore, we restrict the typing rule (poly) by requiring $\sigma_m(F)$ to be of the form $\forall\Delta_m.F(\overline{\tau}) \to \ldots$, where Δ is a sequence of type and functor variables. The restriction to (poly) means that a polytypic definition must define a function, and its first argument, whose type is $F(\overline{\tau})$, has enough information for selecting the appropriate branch of the polytypic definition.

With this restriction the following reduction rules are compatible with the typing (we assume that the constant functors are only $1 : 0$, $\times : 2$ and $+ : 2$):

$$\begin{array}{rcl}
(\text{poly}\ \left[t_1, t_\times, t_+, t^{\Pi}, t^{\circ}, t^{\mu}\right])_0\ (\text{intro}^1) & > & t_1\ (\text{intro}^1) \\
(\text{poly}\ \left[t_1, t_\times, t_+, t^{\Pi}, t^{\circ}, t^{\mu}\right])_2\ (\text{intro}^\times\ t_0\ t_1) & > & t_\times\ (\text{intro}^\times\ t_0\ t_1) \\
(\text{poly}\ \left[t_1, t_\times, t_+, t^{\Pi}, t^{\circ}, t^{\mu}\right])_2\ (\text{intro}^+_j\ t) & > & t_+\ (\text{intro}^+_j\ t) \\
(\text{poly}\ \left[t_1, t_\times, t_+, t^{\Pi}, t^{\circ}, t^{\mu}\right])_m\ (\text{intro}^{\Pi}_{m,i}\ t) & > & t^{\Pi}_{m,i}\ (\text{intro}^{\Pi}_{m,i}\ t) \\
(\text{poly}\ \left[t_1, t_\times, t_+, t^{\Pi}, t^{\circ}, t^{\mu}\right])_m\ (\text{intro}^{\circ}_{n,m}\ t) & > & t^{\circ}_{n,m}\ (\text{intro}^{\circ}_{n,m}\ t) \\
(\text{poly}\ \left[t_1, t_\times, t_+, t^{\Pi}, t^{\circ}, t^{\mu}\right])_m\ (\text{intro}^{\mu}_m\ t) & > & t^{\mu}_m\ (\text{intro}^{\mu}_m\ t)
\end{array}$$

Remark 6.1 The type-theoretic intuition behind the formation rule (poly) is that functors form an inductive family of types indexed by natural numbers. Therefore, one should consider similar formation rules for defining types, functors and type schema by induction on the structure of functors. For instance, definitions of types by induction on the structure of functors are needed to define the type constructor C_F corresponding to the functor expression F by unfolding composition, i.e. $C_{F\langle G_{i:m}\rangle^n}(\overline{X}) = C_F(C_{G_i}(\overline{X})_{i:m})$.

6.3 Polytypic definitions in FML$_{poly}$

FML$_{poly}$ is the extension of FML with the inductive, recursive and polytypic constructions of Sections 6.1.3, 6.1.4 and 6.2. In this section we demonstrate its expressive power by defining the polytypic combinators of FML$_{shape}$. For notational convenience, in the examples we write

$$\begin{array}{ll}
\text{let poly} & \\
\;f_0\ \text{un} & = t_1 \quad | \\
\;f_2\ \langle x_0, x_1\rangle & = t_\times \quad | \\
\;f_2\ (\text{in}_0\ x) & = t_{+,0} \quad | \\
\;f_2\ (\text{in}_1\ x) & = t_{+,1} \quad | \\
\;f_m\ (\text{intro}^\Pi_{m,i}\ x) & = t^\Pi_{m,i} \quad | \\
\;f_m\ (\text{intro}^\circ_{n,m}\ x) & = t^\circ_{n,m} \quad | \\
\;f_m\ (\text{intro}^\mu_m\ x) & = t^\mu_m \quad | \\
\text{in}\ t &
\end{array}$$

for

$$\text{let}\ f = \text{poly}\left[\begin{array}{ll}
\text{elim}^1\ t_1, & \\
\text{elim}^\times\ (\lambda x_0, x_1.t_\times), & \\
\text{elim}^+\ (\lambda x_0.t_{+,0})\ (\lambda x_1.t_{+,1}), & \\
\text{elim}^\Pi_{m,i}(\lambda x.t^\Pi_{m,i}) & i : m : N, \\
\text{elim}^\circ_{n,m}(\lambda x.t^\circ_{n,m}) & m, n : N, \\
\text{match}^\mu_m(\lambda x.t^\mu_m) & m : N
\end{array}\right]\ \text{in}\ t.$$

Remark 6.2 The above notational convention exactly matches the restriction imposed on (poly). There is a minor disuniformity in the definition of the polytypic let, since in all cases we use a destructor, except for μF, where we use the combinator match^μ_m for pattern matching. The reason is that in all other cases destructors are only doing pattern matching.

We show that the polytypic combinators map_m, ltraverse_m and zipop_m are definable in FML$_{poly}$. More precisely, for each polytypic combinator we give a closed term of FML$_{poly}$ such that: it has the same type schema as the combinator, and; the reduction rules for the combinator are derivable.

6.3.1 Definability of map_m

The map_m combinator is defined using an auxiliary combinator, whose only purpose is to rearrange the arguments to fit the restriction imposed by the typing rule (poly).

$$pm \quad : \quad \forall m : N.\forall F : m.\forall X_{i:m}, Y_{i:m} : \mathrm{T}.\ F(\overline{X}) \to (X_i \to Y_i)_{i:m} \to F(\overline{Y})$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mu\ map.\ \text{let poly}} \\
\quad pm_0\ \text{un} & = \text{un} & | \\
\quad pm_2\ \langle x_0, x_1\rangle & = \lambda f_0, f_1.\ \langle f_0\ x_0, f_1\ x_1\rangle & | \\
\quad pm_2\ (\text{in}_0\ x) & = \lambda f_0, f_1.\ \text{in}_0\ (f_0\ x) & | \\
\quad pm_2\ (\text{in}_1\ x) & = \lambda f_0, f_1.\ \text{in}_1\ (f_1\ x) & | \\
\quad pm_m\ (\text{intro}^{\Pi}_{m,i}\ x) & = \lambda f_{i:m}.\ \text{intro}^{\Pi}_{m,i}\ (f_i\ x) & | \\
\quad pm_m\ (\text{intro}^{\circ}_{n,m}\ x) & = \lambda f_{i:m}.\ \text{intro}^{\circ}_{n,m}\ (map_n\ (map_m\ \overline{f})_{j:n}\ x) & | \\
\quad pm_m\ (\text{intro}^{\mu}_{m}\ x) & = \lambda f_{i:m}.\ \text{intro}^{\mu}_{m}\ (map_{m+1}\ \overline{f}\ (map_m\ \overline{f})\ x) & | \\
\quad \multicolumn{3}{l}{\text{in } (\Lambda m.\lambda f_{i:m}.\lambda x.\ pm_m\ x\ \overline{f})}
\end{array}
$$

As an example, here is the derivation of the mapping reduction

$$\text{map}_m\ f_{i:m}\ (\text{intro}^{\mu}_m\ t) \quad > \quad \text{intro}^{\mu}_m\ (\text{map}_{m+1}\ \overline{f}\ (\text{map}_m\ \overline{f})\ t)$$

using the reduction rules for recursive and polytypic definitions. In what follows, we identify map with the term $(\mu\ map. \ldots)$, and pm with its polytypic definition poly pm_0 un $=$ un $\ldots$, in which map has been replaced by its recursive definition.

$$
\begin{array}{ll}
map_m\ f_{i:m}\ (\text{intro}^{\mu}_m\ t) > & (\mu \text{ and let}) \\
(\Lambda m.\lambda f_{i:m}.\lambda x.pm_m\ x\ \overline{f})_m\ \overline{f}\ (\text{intro}^{\mu}_m\ t) > & (\beta) \\
pm_m\ (\text{intro}^{\mu}_m\ t)\ \overline{f} > & (\text{poly and match}^{\mu}_m) \\
(\lambda x.\lambda f_{i:m}.\text{intro}^{\mu}_m\ (map_{m+1}\ \overline{f}\ (map_m\ \overline{f})\ x))\ t\ \overline{f} > & (\beta) \\
\text{intro}^{\mu}_m\ (map_{m+1}\ \overline{f}\ (map_m\ \overline{f})\ t). &
\end{array}
$$

6.3.2 Definability of ltraverse$_m$

The ltraverse$_m$ combinator is defined using an auxiliary combinator

$$
\begin{array}{lll}
plt & : & \forall m : N.\forall F : m.\forall X_{i:m}, Y_{i:m}, S : \mathrm{T}. \\
& & F(\overline{X}) \to (X_i \to P_S(Y_i))_{i:m} \to P_S(F(\overline{Y}))
\end{array}
$$

where P_S is the parsing monad defined in Section 4.2.1.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mu\ lt.\ \text{let poly}} \\
\quad plt_0\ \text{un} & = [\text{un}] & | \\
\quad plt_2\ \langle x_0, x_1\rangle & = \lambda f_0, f_1.\text{let}\, y_0 \Leftarrow f_0\ x_0 \,\text{in let}\, y_1 \Leftarrow f_1\ x_1 \,\text{in}\, [\langle y_0, y_1\rangle] & | \\
\quad plt_2\ (\text{in}_0\ x) & = \lambda f_0, f_1.\text{let}\, y \Leftarrow f_0\ x \,\text{in}\, [\text{in}_0\ y] & | \\
\quad plt_2\ (\text{in}_1\ x) & = \lambda f_0, f_1.\text{let}\, y \Leftarrow f_1\ x \,\text{in}\, [\text{in}_1\ y] & | \\
\quad plt_m\ (\text{intro}^{\Pi}_{m,i}\ x) & = \lambda f_{i:m}.\text{let}\, y \Leftarrow f_i\ x \,\text{in}\, [\text{intro}^{\Pi}_{m,i}\ y] & | \\
\quad plt_m\ (\text{intro}^{\circ}_{n,m}\ x) & = \lambda f_{i:m}.\text{let}\, y \Leftarrow lt_n (lt_m\ \overline{f})_{j:n}\ x \,\text{in}\, [\text{intro}^{\circ}_{n,m}\ y] & | \\
\quad plt_m\ (\text{intro}^{\mu}_{m}\ x) & = \lambda f_{i:m}.\text{let}\, y \Leftarrow lt_{m+1}\ \overline{f}\ (lt_m\ \overline{f})\ x \,\text{in}\, [\text{intro}^{\mu}_{m}\ y] & | \\
\quad \multicolumn{3}{l}{\text{in } (\Lambda m.\lambda f_{i:m}.\lambda x.plt_m\ x\ \overline{f})}
\end{array}
$$

6.3.3 Definability of zipop$_m$

The zipop$_m$ combinator is defined using an auxiliary combinator

$$
\begin{array}{lll}
pz & : & \forall m : N.\forall F : m.\forall X_{i:m}, Y_{i:m}, Z_{i:m}, S : \mathrm{T}. \\
& & F(\overline{X}) \to (X_i \to Y_i \to P_S(Z_i))_{i:m} \to F(\overline{Y}) \to P_S(F(\overline{Z}))
\end{array}
$$

where P_S is the parsing monad defined in Section 4.2.1.

$$
\begin{array}{l}
\mu\ zop.\ \mathrm{let\ poly} \\
\quad \begin{array}{lcl}
pz_0\ \mathrm{un} & = & \mathrm{elim}^1\ ([\mathrm{un}]) \qquad | \\
pz_2\ \langle x_0, x_1\rangle & = & \begin{array}{l}\lambda f_0, f_1.\mathrm{elim}^{\times}(\lambda y_0, y_1. \\ \mathrm{let}\ z_0 \Leftarrow f_0\ x_0\ y_0\ \mathrm{in\ let}\ z_1 \Leftarrow f_1\ x_1\ y_1\ \mathrm{in}\ [\langle z_0, z_1\rangle])\end{array} \qquad | \\
pz_2\ (\mathrm{in}_0\ x) & = & \lambda f_0, f_1.\mathrm{elim}^{+}\ (\lambda y.\mathrm{let}\ z \Leftarrow f_0\ x\ y\ \mathrm{in}\ [\mathrm{in}_0\ z])\ (\lambda y.\mathrm{fail}) \ | \\
pz_2\ (\mathrm{in}_1\ x) & = & \lambda f_0, f_1.\mathrm{elim}^{+}\ (\lambda y.\mathrm{fail})\ (\lambda y.\mathrm{let}\ z \Leftarrow f_1\ x\ y\ \mathrm{in}\ [\mathrm{in}_1\ z]) \ | \\
pz_m\ (\mathrm{intro}^{\Pi}_{m,i}\ x) & = & \lambda f_{i:m}.\mathrm{elim}^{\Pi}_{m,i}\ (\lambda y.\mathrm{let}\ z \Leftarrow f_i\ x\ y\ \mathrm{in}\ [\mathrm{intro}^{\Pi}_{m,i}\ z]) \qquad | \\
pz_m\ (\mathrm{intro}^{\circ}_{n,m}\ x) & = & \begin{array}{l}\lambda f_{i:m}.\mathrm{elim}^{\circ}_{n,m}(\lambda y. \\ \mathrm{let}\ z \Leftarrow zop_n(zop_m\ \overline{f})_{j:n}\ x\ y\ \mathrm{in}\ [\mathrm{intro}^{\circ}_{n,m}\ z])\end{array} \qquad | \\
pz_m\ (\mathrm{intro}^{\mu}_{m}\ x) & = & \begin{array}{l}\lambda f_{i:m}.\mathrm{elim}^{\mu}_{m}(\lambda y. \\ \mathrm{let}\ z \Leftarrow zop_{m+1}\ \overline{f}\ (zop_m\ \overline{f})\ x\ y\ \mathrm{in}\ [\mathrm{intro}^{\mu}_{m}\ z])\end{array} \qquad |
\end{array} \\
\quad \mathrm{in}\ (\Lambda m.\lambda f_{i:m}.\lambda x, y.pz_m\ x\ \overline{f}\ y)
\end{array}
$$

6.3.4 Limitations of FML_{poly} *à la Curry*

We give some examples of polytypic functions that are definable in FML_{poly} with the unrestricted (poly) rule, but cannot be defined using the restricted (poly) rule, where $\sigma_m(F) \equiv \forall\Delta.F(\overline{\tau}) \to \ldots$ and Δ is a sequence of type and functor variables. These examples identify clearly the limitations at the level of polytypic definitions, when terms have no explicit type and functor information.

Example 6.3 Consider the polytypic function $is_C : \forall m : N.\forall F : m.Bool$, which tests whether F is equal to C. Clearly, we cannot define it using the restricted (poly) rule. However, we could add an extra argument of type $F(\overline{1})$, so that the restricted (poly) rule become applicable and we have enough information about F.

This trick does not get very far, e.g. consider $occur_C : \forall m : N.\forall F : m.Bool$, which tests whether C occurs in F. In this case we cannot get the necessary information about F by inspecting a term of type $F(\overline{1})$, e.g. when $C = 1$ and F is the functor $F(X) = X + 1$, more formally $F = +\langle \Pi^1_1, 1\langle\rangle^1\rangle^1$.

Example 6.4 A more interesting example is the problem of constructing polytypic parser, which takes a string and produces an element of type $F(\overline{X})$. This parser is used in the polytypic data compression algorithm of (Jansson & Jeuring, 1997; Jansson, 1997). More precisely, we would like a combinator for generating a parser for the type $F(\overline{X})$ given parsers for the X_i

$$parse \quad : \quad \forall m : N.\forall F : m.\forall X_{i:m} : \mathrm{T}.(P_S X_i)_{i:m} \to P_S(F(\overline{X}))$$

where P_S is the parsing monad $P_S X = S \to (X \times S) + 1$ defined in Section 4.2.1, and usually the type parameter S consists of either lists of characters or lists of tokens. (Björk, 1997) define a more complex *parse* combinator

```
parse :: (d a -> Struct a) -> Parser String a -> Parser String (d a)
```

parameterized over a regular datatype d rather than a functor.

It is natural to decompose *parse* in two parts: *parseshape* for parsing the shape information, and *parsedata* for filling the shape with data.

$$\begin{array}{rcl} parseshape & : & \forall m : N.\forall F : m.P_S(F(\overline{1})) \\ parsedata & : & \forall m : N.\forall F : m.\forall X_{i:m} : \mathrm{T}.(P_S X_i)_{i:m} \to F(\overline{1}) \to P_S(F(\overline{X})) \end{array}$$

parsedata is unproblematic and could be defined as an instance of $\mathrm{ltraverse}_m$. Defining *parseshape* requires either explicit knowledge of the functor F or a single type to describe all possible shapes.

7 Comparison of FML with PolyP

In this section we make a comparison between **PolyP** (as described in (Jansson & Jeuring, 1997)) and FML. At present FML is only a formal calculus, where choices and issues are spelled out and addressed precisely, but the language has not been implemented. On the other hand, **PolyP** is being implemented as an extension of *Haskell*, but some issues are resolved in a crude way in order to get an implementation up and running, and less attention has been give to a rigorous documentation.

Functors and datatype constructors. FML starts from the observation that the categorical concept of a functor is an abstraction with which it is both useful and feasible to write programs. Indeed several researchers have advocated *categorical programming* and stressed the importance of functors (e.g. (Hagino, 1987a; Hagino, 1987b; Meijer *et al.*, 1991; Cockett & Fukushima, 1992; Cockett & Spencer, 1995)).

In FML, functors are considered as fundamental as types (and type schema); there is no attempt to explain them in terms of type constructors with additional structure and properties. Its functor application is the canonical way of getting a type constructor from a functor, but there is no way to go in the opposite direction.

On the contrary, in **PolyP** functors (of arity 2) seem auxiliary to *regular* datatypes (of arity 1). The **PolyP** operations

$$\begin{array}{rcl} FunctorOf & : & Regular \to Bifunctor \\ Mu & : & (Bifunctor, Regular) \to Regular \end{array}$$

do not have a clear category-theoretic reading (unlike the FML functor constructors). They appear to be motivated mainly by the desire to keep information about names of datatype constructors in the qualified kind *Regular*, but ignore it in *Bifunctor*. In particular, *FunctorOf* is defined by induction on the syntax of datatype definitions. For example, given the datatypes

$$\begin{array}{rcl} Empty_1\ X & = & \text{empty of } Empty_1\ X \\ Empty_2\ X & = & \text{empty of } Empty_1\ X \end{array}$$

$FunctorOf(Empty_1) = Rec$ and $FunctorOf(Empty_2) = Empty_1@Par$ are different, although the types $Empty_1\ X$ and $Empty_2\ X$ are isomorphic to the empty type. *FunctorOf* does not make much sense semantically because it recovers a functor from its initial algebra.

Functor arities. At present **PolyP** handles only bifunctors, i.e. functors of arity 2. Moreover, the syntactic category of bifunctors F makes reference to two other syntactic categories: datatypes D and types τ. These are not given explicitly, thus it is unclear whether they correspond to the FML arities 1 and 0.

Fixing an upper bound to functor arities is a simple way to avoid both the infinitary rules (that we adopted in the informal account of FML_{poly}) and the dependent types (that are needed in a formal account). Such upper bounds are inherently ad hoc, but once the upper bound is chosen an implementation of FML_{poly} becomes feasible.

Separation between functors and types. FML makes a clear decision: the language for functors comes before and is independent from the language for types. This is very important for having a considerable degree of freedom in interpreting functors (see Section 2.5). On the other hand, **PolyP** has a construct $Con : * \to Bifunctor$, which converts a type into a constant bifunctor. Because of this the bifunctors of **PolyP** may fail to preserve equality types. Moreover, it is unclear how the $Con\ \tau$ case of a polytypic definition should be handled, i.e. whether it may treat different τs differently or it must treat them uniformly.

Type inference. Type inference for FML is a fairly straightforward extension of type inference for ML. In **PolyP** the situation is more delicate. In the definition of a polytypic function the type schema should be given explicitly. However, Section 2.2 of (Jansson & Jeuring, 1997) claims that one can infer the functor argument of a polytypic function. This seems doubtful without restrictions on the type schema allowed in polytypic definitions, like those introduced in the typing rule (poly) of FML_{poly}. For instance, we expect type inference to be problematic, when one allows polytypic functions like $parse$ (see Example 6.4).

Inductive functors or beyond. The polytypic construct of **PolyP** is very natural and convenient for dealing with inductive functors. Indeed in FML_{poly} one may easily define polytypic combinators which have to be taken as primitive in FML_{basic} and FML_{shape}. On the other hand, FML_{basic} and FML_{shape} take an open-ended view of functors, which is incompatible with mechanisms allowing the programmer to define their own polytypic operations by induction on the structure of functors. In these calculi the only way to build polytypic functions is by combining existing ones according to fixed patterns. Nevertheless the combinators of FML_{shape} are powerful enough to express algorithms regarded as motivating examples for the polytypic construct. The intended semantic of FML_{shape} tell us that its polytypic combinators are applicable to a larger class of functors than the inductive ones. So there is not a clear best choice between FML_{shape} and FML_{poly}.

8 Conclusions

FML is an extension of the Hindley-Milner type system that supports parametric functorial polymorphism. That is, one can write algorithms (e.g. mapping and folding) which work uniformly for a large class of functorial type constructors, unlike previous, ad hoc algorithms. The Hindley-Milner type inference algorithm extends smoothly to FML and reduction on well-formed terms is confluent and strongly normalizing.

The functor syntax admits functors of many variables, and functor composition. Canonical isomorphisms are used to distinguish different orders of composition, which allow terms to express the shape-data, or functor-argument decomposition necessary to locate their data. This feature is essential for parametric algorithms.

Much remains to be done. We expect that the usual denotational models of system F can be extended to handle explicit functors. Also, the exact relationship between FML and Fω is not yet clear. Many of the subscripts on the combinators seem to be redundant. By introducing *form variables* (Jay, 1995a) to represent sequences of types we may be able to infer many of them, just as we infer types. Another, basic shape polymorphic operation is that of extracting the data from the shape. This is fundamental to search operations, pattern-matching etc. and should be comfortably supported within the current system, as a new combinator.

FML should be considered as an intermediate language. Indeed, the examples show that FML is rather awkward in comparison with ML. FML provides a fine analysis of access to data via the canonical isomorphisms, and should be compared with other intermediate languages, such as those proposed in (Peyton Jones, 1991; Leroy, 1992) to distinguish between boxed and unboxed values and providing explicit coercions between them. One can envisage an intensional semantics where $\mathrm{intro}^{\Pi}_{m,i}(t) \in \Pi^m_i(\overline{X})$ is like a boxed value, since t is wrapped with additional information about m and i, while $\mathrm{intro}^{\circ}_{m,n}$ acts like data redistribution. Here, distinguishing between types and functors is crucial.

Another possibility is to add additional base functors that are not inductive, e.g. arrays. This would allow for types such as trees whose nodes support arrays, as appeared in (Wu *et al.*, 1997), and connect to research on shape analysis (Jay & Sekanina, 1997; Jay *et al.*, 1997; Jay & Steckler, 1998).

Finally, it remains to implement FML as an extension of an existing programming language, so that its merits can be tested by the community of programmers.

References

Aczel, P. (1978). *A general Church-Rosser theorem.* Tech. rept. Univ. of Manchester.

Aditya, S., Flood, C., & Hicks, J. 1994 (June). Garbage collection for strongly-typed languages using run-time type reconstruction. *Pages 12–23 of: ACM conference on lisp and functional programming.*

Bainbridge, E.S., Freyd, P.J., Scedrov, A., & Scott, P.J. (1990). Functorial polymorphism. *Theoretical computer science*, **70**, 35–64.

Barendregt, H.P. (1984). *The lambda calculus: Its syntax and semantics.* North Holland. revised edition.

Barendregt, H.P. (1992). Lambda calculi with types. *Handbook of logic in computer science.* Oxford Univ. Press.

Barr, M., & Wells, C. (1990). *Category theory for computing science.* International Series in Computer Science. Prentice Hall.

Bellè, G., Jay, B., & Moggi, E. (1996). Functorial ML. *PLIPL'96*. LNCS, vol. 1140. Springer Verlag.

Bellè, G., Jay, C. B., & Moggi, E. (1998). *Functorial ML (including appendices)*. Tech. rept. DISI, Univ. of Genova. DISI-TR-98-03.

Benabou, J. (1967). *Introduction to bicategories*. Lecture Notes in Mathematics, vol. 47. Springer.

Björk, Staffan L. (1997). *Parsers, Pretty Printers and PolyP*. M.Phil. thesis, Göteborg University.

Cockett, J. Robin B., & Spencer, Dwight. (1995). Strong categorical datatypes II: A term logic for categorical programming. *Theoretical computer science*, **139**(1–2), 69–113.

Cockett, J.R.B. (1990). List-arithmetic distributive categories: locoi. *Journal of pure and applied algebra*, **66**, 1–29.

Cockett, J.R.B., & Fukushima, T. (1992). *About* **charity**. Tech. rept. 92/480/18. University of Calgary.

Girard, J.-Y., Lafont, Y., & Taylor, P. (1989). *Proofs and Types*. Cambridge Tracts in TCS, vol. 7. Cambridge University Press.

Goguen, J.A., Thatcher, J.W., Wagner, E.G., & Wright, J.B. (1977). Initial algebra semantics and continuous algebras. *Journal of the acm*, **24**, 68–95.

Hagino, T. (1987a). *A categorical programming language*. Ph.D. thesis, University of Edinburgh.

Hagino, T. (1987b). A typed lambda calculus with categorical type constructors. *Pages 140–157 of:* Pitt, D. H., Poigné, A, & Rydeheard, D. E. (eds), *Category and Computer Science*. Lecture Notes in Computer Science, vol. 283. Springer.

Harper, R., & Morrisett, G. 1995 (January). Compiling polymorphism using intensional type analysis. *Pages 130–141 of: Conference record of popl '95: 22nd acm sigplan-sigact symposium on principles of programming languages.*

Hook, James, & Sheard, Tim. 1993 (November). *A semantics of compile-time reflection*. Tech. rept. 93-019. Department of Computer Science and Engineering, Oregon Graduate Institute.

Jansson, Patrik. (1997). *Functional polytypic programming — use and implementation*. Tech. rept. Chalmers University of Technology, Gothenburg, Sweden. Licentiate thesis. Available from `http://www.cs.chalmers.se/~patrikj/lic/`.

Jansson, Patrik, & Jeuring, Johan. (1997). PolyP – a polytypic programming language extension. *Pages 470–482 of: Conference record of POPL'97: the 24th ACM sigplan-sigact symposium on principles of programming languages.*

Jay, C.B. (1995a). Polynomial polymorphism. *Pages 237–243 of:* Kotagiri, R. (ed), *Proceedings of the eighteenth Australasian computer science conference: Glenelg, South Australia 1–3 February, 1995*, vol. 17. A.C.S. Communications.

Jay, C.B. (1995b). A semantics for shape. *Science of computer programming*, **25**, 251–283.

Jay, C.B. (1995c). Shape analysis for parallel computing. *Pages 287–298 of:* Darlington, J. (ed), *Proceedings of the fourth international parallel computing workshop: Imperial College London, 25–26 September, 1995.* Imperial College/Fujitsu Parallel Computing Research Centre.

Jay, C.B. (1997). Covariant types. *Theoretical computer science*, **185**, 237–258.

Jay, C.B., & Sekanina, M. (1997). Shape checking of array programs. *Pages 113–121 of: Computing: the Australasian theory seminar, proceedings, 1997.* Australian Computer Science Communications, vol. 19.

Jay, C.B., & Steckler, P.A. (1998). The functional imperative: shape! Hankin, C.L. (ed), *Proceedings ESOP'98*. to appear.

Jay, C.B., Cole, M.I., Sekanina, M., & Steckler, P. (1997). A monadic calculus for parallel costing of a functional language of arrays. *Pages 650–661 of:* Lengauer, C., Griebl, M., & Gorlatch, S. (eds), *Euro-par'97 parallel processing*. Lecture Notes in Computer Science, vol. 1300. Springer.

Jeuring, J. (1995). Polytypic pattern matching. *Pages 238–248 of: Conference on functional programming languages and computer architecture.*

Jeuring, J., & Jansson, P. (1996). Polytypic programming. *Pages 68–114 of:* Launchbury, J., Meijer, E., & Sheard, T. (eds), *Advanced Functional Programming, Second International School*. Springer-Verlag. LNCS 1129.

Jones, M.P. (1995). A system of constructor classes: overloading and implicit higher-order polymorphism. *J. of functional programming*, **5**(1).

Joyal, A. (1981). Une théorie combinatoire des séries formelles. *Advances in mathematics*, **42**, 1–82.

Kfoury, A. J., & Tiuryn, J. (1992). Type reconstruction in finite rank fragments of the second-orderλ-calculus. *Information and computation*, **98**(2), 228–257.

Klop, J.W. (1980). *Combinatory reduction systems*. Ph.D. thesis, Mathematical Center Amsterdam. Tracts 129.

Lambek, J., & Scott, P.J. (1986). *Introduction to higher-order categorical logic*. Cambridge Studies in Advanced Mathematics, vol. 7. Cambridge University Press.

Leroy, X. (1992). Unboxed objects and polymorphic typing. *19th symposium on principles of programming languages*. ACM Press.

MacLane, S. (1971). *Categories for the working mathematician*. Springer Verlag.

Meertens, L. (1996). Calculate polytypically! *Pages 1–16 of:* Kuchen, & Swierstra (eds), *Programming Languages: Implementations, Logics, and Programs*. Lecture Notes in Computer Science, vol. 1140. Springer-Verlag. invited talk.

Meijer, E., & Hutton, G. (1995). Bananas in space: extending fold and unfold to exponential types. *Procedings 7th international conference on functional programming and computer architecture, San Diego, California, June 1995*. ACM Press.

Meijer, E., Fokkinga, M., & Paterson, R. (1991). Functional programming with bananas, lenses, envelopes and barbed wire. *Pages 124–44 of:* Hughes, J. (ed), *Procceding of the 5th ACM conference on functional programming and compter architecture*. Lecture Notes in Computer Science, vol. 523. Springer Verlag.

Mendler, N.P. (1991). Inductive types and type constraints in the second-order lambda calculus. *Annals of pure and applied logic*, **51**.

Milner, R. (1978). A theory of type polymorphism in programming. *Jcss*, **17**.

Moggi, E. (1991). Notions of computation and monads. *Information and computation*, **93**(1).

Nordström, Bengt, Petersson, Kent, & Smith, Jan M. (1990). *Programming in Martin-Löf's type theory:an introduction*. International series of monographs on computer science, no. 7. Oxford University Press, New York.

Peyton Jones, S. (1991). Unboxed values as first-class citizens. *Functional programming and computer architecture*. LNCS, vol. 523.

Reynolds, J., & Plotkin, G.D. (1990). On functors expressible in polymorphic lambda-calculus. Huet, G. (ed), *Logical foundations of functional programming*. Addison-Wesley.

Sheard, Tim. 1993 (November). *Type parametric programming*. Tech. rept. 93-018. Department of Computer Science and Engineering, Oregon Graduate Institute.

Tofte, M. (1988). *Operational semantics and polymorphic type inference*. Ph.D. thesis, University of Edinburgh. available as CST-52-88.

Tolmach, A. 1994 (June). Tag-free garbage collection using explicit type parameters. *Pages 1–11 of: ACM conference on lisp and functional programming.*

Wadler, P. (1993). Comprehending monads. *Mathematical structures in computer science.* Special issue of selected papers from 6'th Conference on Lisp and Functional Programming, 1992.

Wu, Q., Field, A.J., & Kelly, P.H.J. (1997). M-tree: a parallel abstract data type for block-irregular adaptive applications. *Pages 638–49 of:* Lengauer, C., Griebl, M., & Gorlatch, S. (eds), *Euro-par'97 parallel processing.* Lecture Notes in Computer Science, vol. 1300. Springer.

Contents

1 Introduction 2
2 Functors, types and type schema 5
2.1 Functors 5
2.2 Syntax 6
2.3 Examples 8
2.4 Substitutions and unification 9
2.5 Semantics 10
3 Basic terms 15
3.1 Syntax 15
3.2 Type inference algorithm 18
4 Constants and Reductions 19
4.1 Basic constants and FML_{basic} 20
4.2 Syntactic sugar 21
4.3 Shape-based constants and FML_{shape} 23
4.4 Properties of reduction 26
5 Extended examples 26
5.1 One-side matching 27
5.2 Unification 28
6 Variations and extensions to FML 34
6.1 Type-theoretic variations and extensions 36
6.2 Polytypic definitions 38
6.3 Polytypic definitions in FML_{poly} 40
7 Comparison of FML **with PolyP** 43
8 Conclusions 44
References 45