Functorial ML

G. Bellè[2], C.B. Jay[1] and E. Moggi[2]

[1] SOCS - Univ. of Tech. Sydney, P.O. Box 123 Broadway, 2007, Australia
phone: ++612 330-1814, fax: ++612 330-1807, e-mail: cbj@socs.uts.edu.au
[2] DISI - Univ. di Genova, via Dodecaneso 35, 16146 Genova, Italy
phone: +39 10 353-6629, fax: +39 10 353-6699, e-mail: {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 normalising, and supports type inference.

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 [LS86] and the description of polymorphic terms as natural transformations (e.g. [BFSS90]). For example, the operation of appending lists can be represented as a natural transformation $L \times L \Rightarrow L$ where $L\colon \mathcal{D} \to \mathcal{D}$ 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 [RP90], but such encodings are rather unsatisfactory ([GLT89, Section 15.1.1]). In particular, the action of a functor on morphism, 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. [MFP91, MH95, Jeu95]), is to give a combinator for mapping, whose type can be expressed as:

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

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

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

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 realising this high-level algorithm, since the question of how to *find* the data remains.

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* [Jon95]. 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 [CF92] encodes this directly. Jeuring [Jeu95] uses a pre-processor to determine the appropriate Haskell code for mapping and other *polytypic* operations. *Intensional polymorphism* [HM95] 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** [Jay95a]. Polymorphic mapping for an extension of the *covariant* type system [Jay96] was produced in [Jay]. This paper presents an extension of Hindley-Milner called Functorial ML, or FML, which supports parametric functorial polymorphism.

For example, it supports:

$$\begin{aligned} \text{map}\ f\ (\text{cons}\ h\ t) &\rightarrow_* \text{cons}\ (f\ h)\ (\text{map}\ f\ t) \\ \text{map}\ f\ (\text{leaf}\ x) &\rightarrow_* \text{leaf}\ (f\ x) \end{aligned}$$

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 combinators, 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:

$$\text{let}\ g = \text{map}\ f\ \text{in pair}\ (g\ (\text{cons}\ h\ t)\ (g\ (\text{leaf}\ x)))$$

where pair 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

$$\begin{aligned} \text{map}_2\ f\ g\ (\text{in}_0\ t) &\rightarrow \text{in}_0\ (f\ t) & (1) \\ \text{map}_2\ f\ g\ (\text{pair}\ s\ t) &\rightarrow \text{pair}\ (f\ s)\ (g\ t)\ . & (2) \end{aligned}$$

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

Shape theory [Jay95b] 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 $\Pi^m_i\colon \mathcal{D}^m \to \mathcal{D}$ which pick out the ith argument from m. It is tempting to identify $\Pi^2_0(X,X)$ and $\Pi^2_1(X,X)$ with X but this would obliterate the shape-data distinction. Rather, these types are "isomorphic", a situation captured by terms

$$\mathrm{pex}_{2,j}\colon X_j \to \Pi^2_j(X_0, X_1)$$

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

$$\begin{aligned}\mathrm{map}_2\ f\ g\ (\mathrm{pex}_{2,0}\ x) &\to \mathrm{pex}_{2,0}\ (f\ x)\\ \mathrm{map}_2\ f\ g\ (\mathrm{pex}_{2,1}\ x) &\to \mathrm{pex}_{2,1}\ (g\ x)\ .\end{aligned}$$

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.

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

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:

$$\mathrm{map}\ f\ (\mathrm{dex}_{1,1}\ t) \to \mathrm{dex}_{1,1}\ (\mathrm{map}\ (\mathrm{map}\ f)\ t)$$

whose corresponding diagram is:

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

These isomorphisms may be viewed as a systematic method for resolving the ambiguities 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 [Jay95c] in which case the isomorphisms represent redistributions of the data. Also, such isomorphisms between different composites are central to bicategories [Ben67].

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 schemes (so extending system F with functors). Such a system is used in the proof of strong normalisation. Again, one could identify types with functors of arity 0, at the cost of introducing another pair of isomorphisms. We emphasise the tri-partite division for reasons of clarity, and to obtain type inference.

In this paper we will consider only FML with untyped terms, i.e. à la Church (in the terminology of [Bar92]), because the main focus is on parametric functorial polymorphism. However, FML à la Curry is very important, too: it allows a more aggressive use of type information (as advocated in [HM95]) and it is more suitable for a semantic investigation.

The following sections of the paper are devoted to: the type system (functors, types and type schema); terms and type assignment, including the type inference algorithm; term reduction and its properties (subject reduction, Church-Rosser and strong normalization); examples, and conclusions and future work. For more technical details the reader can refer to [BJM96].

2 Functors, types and type schema

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 symbols m and n will denote natural numbers throughout this paper. If $F\colon \mathcal{D}^m \to \mathcal{D}$ is a functor from the mth power of a category $\mathcal{D}$ to itself, then we may express this by saying F is of arity m and write it as $F\colon m$.

Here are some elementary examples and constructions. $+, \times\colon 2$ are binary functors representing sums and products, and $1\colon 0$ is the functor of no arguments that produces the terminal object, corresponding to the unit type. Let i range over $m = \{0, \ldots, m-1\}$. The ith projection functor of m arguments is $\Pi_i^m\colon m$. A sequence $G_i\colon n$ of functors may be written as $G_{i\in m}$ 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 $F\colon m$ then

$$\begin{tikzcd} \mathcal{D}^n \arrow[r, "\langle\overline{G}\rangle"] & \mathcal{D}^m \arrow[r, "F"] & \mathcal{D} \end{tikzcd}$$

is their *composite*. If $F\colon m+1$ is a functor then $\mu^m F\colon m$ represents the functor whose action on the tuple $\overline{X}$ yields the initial $F(\overline{X}, -)$-algebra (e.g. [BW90])

$$F(\overline{X}, \mu^m F(\overline{X})) \to \mu^m F(\overline{X})$$

used to find minimal solutions of recursive domain equations.

These constructions motivate the choice of functors in the following description of the raw syntax for functors, types and type schema.

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

We adopt the following notational conventions throughout. X and Y range over functor and type variables. C ranges over functor constants, in particular we will consider $+, \times$, the binary functors representing sums and products, and 1, the functor of no arguments that produces the terminal object, corresponding to the unit type. F and G range over functors (though sometimes are used as functor variables), τ ranges over types, and σ ranges over type schema throughout the paper. A type τ may be distinguished from functors or type schema by giving it the fixed arity $\tau\colon \mathrm{T}$. 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 $1()$.

The functor notation is as described above. The given types are variables, functor applications, and function types. The application of a functor to a tuple $\tau_{i\in m}$ of types represents the action of the categorical functor on objects. Note that all of the ancillary type constructors, such as products and sums, have been pushed into the declaration of the functors.

The type schema are types, universal quantification over type variables and universal quantification 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.

A *type context* (notation Δ) is a sequence of type variables with assigned arities (either $X\colon n$ or $X\colon \mathrm{T}$) 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.

For each of the syntactic categories above we give rules to infer when a raw expression of that category is well-formed in a type context. The symbol i ranges over m. $\Delta \vdash$ means that Δ is a well-formed typed context.

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

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

$\Delta \vdash F\colon n$ means that F is a functor of arity n in context Δ. The formation rules for functors express the constraints on arities implicit in the category theory.

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

$$\text{(FPi)}\ \frac{\Delta \vdash}{\Delta \vdash \Pi_i^m\colon m} \qquad \text{(Fcomp)}\ \frac{\Delta \vdash F\colon m \quad \Delta \vdash G_i\colon n}{\Delta \vdash F\langle G_{i\in m}\rangle^n\colon n}$$

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

$\Delta \vdash \tau{:}\,\mathrm{T}$ means that τ is a type in context Δ.

$$(X)\ \frac{\Delta \vdash}{\Delta \vdash X{:}\,\mathrm{T}}\ \mathrm{T} = \Delta(X) \qquad (\mathrm{Fapp})\ \frac{\Delta \vdash F{:}\, m \quad \Delta \vdash \tau_i{:}\,\mathrm{T}}{\Delta \vdash F(\tau_{i\in m}){:}\,\mathrm{T}}$$

$$(\to)\ \frac{\Delta \vdash \tau_1, \tau_2{:}\,\mathrm{T}}{\Delta \vdash \tau_1 \to \tau_2{:}\,\mathrm{T}}$$

$\Delta \vdash \sigma$ means that σ is a type schema in context Δ.

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

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

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. We denote with $\forall\Delta.\tau$ the following schema: τ, if $\Delta = \emptyset$, $\forall\Delta'.(\forall X{:}\, n.\tau)$ if $\Delta = \Delta', X{:}\, n$ and $\forall\Delta'.(\forall X{:}\,\mathrm{T}.\tau)$ if $\Delta = \Delta', X{:}\,\mathrm{T}$.

Lemma 1. *1. Uniqueness of derivation: each judgement $\Delta \vdash J$ 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. □

A *substitution* is a partial function S from type variables to expressions for functors, types or schema. The action of a substitution is extended homomorphically to any expressions containing free type variables. If R is another substitution then their *composite* substitution $S\ R$ has action given by $(S\ R)X = S(RX)$. The notation $S{:}\,\Delta_1 \to \Delta_2$ means that 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

$$\Delta_2 \vdash SX{:}\,\Delta_1(X)\ .$$

S is a *renaming* if it is an injective function from variables to variables. Then define Δ_S by

$$\emptyset_S = \emptyset$$
$$(\Delta, X{:}\, a)_S = \Delta_S, S(X){:}\, a\ .$$

Lemma 2.

1. *Renaming: let S be 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: let $S{:}\,\Delta_1 \to \Delta_2$ be 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. The first two results are used to handle the $\forall$ rules in the latter proofs. □

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 $S' = R\ S$ on $\mathrm{DV}(\Delta)$.

Lemma 3. *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 lead to higher order unification since, for example, $\Pi_i^m(\overline{X})$ and X_i do not have a unifier. □

3 Terms and type assignment

The Hindley-Milner type system may assign either type schema or types to terms [Tof88]). The former has separate rules for abstracting and instantiating type variables, whereas the latter combines these with the rules for typing variables and combinators, and the let-construct, respectively. Both type assignment systems can be extended easily to FML. Here, we consider only the latter, since it is closer to the type inference algorithm.

The untyped terms we will consider here are like those considered in Hindley-Milner, but with several additional constants and no fix-point combinator (since we wish to have strong normalisation):

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

These terms are variables; constants; λ-abstractions and applications and a let-construct. The novelty, and power, of the system resides in the more powerful types assigned to these terms, and the choice of constants, which will be used to capture important properties of functors.

Their description uses the following notational conventions which 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*, i.e. sequences of $x{:}\,\sigma$ with no repetitions of term variables x. The usual conventions of λ-calculus concerning grouping of declared and bound variables apply (see [Bar84]); $\mathrm{FV}(t)$ is the set of free variables of t; $e'\{e/x\}$ is the substitution of e for x in e'.

Here are the term formation rules. $\Delta;\Gamma\vdash$ means that $\Delta;\Gamma$ is a well-formed context.

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

$\Delta;\Gamma\vdash t{:}\,\tau$ means that t is a term of type τ in context $\Delta;\Gamma$ and hence that t is *typable* in this context.

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

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

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

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

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

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 $\mathrm{DV}(\Gamma_1)$ is included in the domain of S and that $\Delta_2,\Delta;\Gamma_2\vdash S(x){:}\,S(\tau)$ 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 4.

1. *$\Delta;\Gamma\vdash$ implies $\Delta\vdash\Gamma(x)$ for any $x\in\mathrm{DV}(\Delta)$.*
2. *Well-typing: $\Delta;\Gamma\vdash t{:}\,\tau$ implies $\Delta\vdash\tau{:}\,\mathrm{T}$.*
3. *Let $S{:}\,\Delta_1\to\Delta_2$ be a substitution; then $\Delta_1;\Gamma\vdash$ implies $\Delta_2;S(\Gamma)\vdash$.*
4. *Type substitution: let $S{:}\,\Delta_1\to\Delta_2$ be a substitution; then $\Delta_1;\Gamma\vdash t{:}\,\tau$ implies $\Delta_2;S(\Gamma)\vdash t{:}\,S(\tau)$.*
5. *Let S be a renaming of Δ; then $\Delta;\Gamma\vdash J$ implies $\Delta_S;\Gamma_S\vdash S(J)$.*
6. *Thinning: $\Delta;\Gamma_1,\Gamma_2\vdash J$ implies $\Delta;\Gamma_1,x{:}\,\sigma,\Gamma_2\vdash J$ for any $x\notin\mathrm{DV}(\Gamma_1,\Gamma_2)$.*
7. *Term substitution: let $\Delta_1,\Delta;\Gamma_1\vdash t{:}\,\tau$; 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 combinators are of two kinds. The first collection express properties of the functorial calculus. The others capture properties of the functor constants introduced to the basic system. The symbol i ranges over m and j ranges over 2. In the first group we have:

$$\begin{aligned}
\mathrm{map}_m &: \forall F\colon m.\forall X_{i\in m}, Y_{i\in m}\colon \mathrm{T}.(X_i \to Y_i) \to_{i\in m} F(\overline{X}) \to F(\overline{Y}) \\
\mathrm{pex}_{m,i} &: \forall X_{j\in m}\colon \mathrm{T}.X_i \to \Pi^m_i(\overline{X}) \\
\mathrm{pin}_{m,i} &: \forall X_{j\in m}\colon \mathrm{T}.\Pi^m_i(\overline{X}) \to X_i \\
\mathrm{dex}_{m,n} &: \forall F\colon m.\forall G_{i\in m}\colon n.\forall X_{j\in n}\colon \mathrm{T}.F(G_i(\overline{X})_{i\in m}) \to F\langle\overline{G}\rangle^n(\overline{X}) \\
\mathrm{din}_{m,n} &: \forall F\colon m.\forall G_{i\in m}\colon n.\forall X_{j\in n}\colon \mathrm{T}.F\langle\overline{G}\rangle^n(\overline{X}) \to F(G_i(\overline{X})_{i\in m}) \\
\mathrm{intro}_m &: \forall F\colon m+1.\forall X_{i\in m}\colon \mathrm{T}.F(\overline{X}, \mu^m F(\overline{X})) \to \mu^m F(\overline{X}) \\
\mathrm{fold}_m &: \forall F\colon m+1.\forall X_{i\in m}, Y\colon \mathrm{T}.(F(\overline{X}, Y) \to Y) \to \mu^m F(\overline{X}) \to Y\ .
\end{aligned}$$

map_m expresses the action of functors of arity m on m-tuples of morphisms. The rest of the combinators in this group are linked to the various functor formation rules. The pairs of terms $\mathrm{pex}_{m,i}$ and $\mathrm{pin}_{m,i}$, and $\mathrm{dex}_{m,n}$ and $\mathrm{din}_{m,n}$ should be thought of as pairs of inverse isomorphisms. $\mathrm{pex}_{m,i}$ makes its argument the ith argument of m. It is used to store data in a uniform way, suitable for mapping. $\mathrm{dex}_{m,n}$ is an isomorphism between two different ways of associating a triple composition of functors. intro_m and fold_m are the introduction and elimination terms for the initial algebra functors. Recalling that initial algebras are defined for a functor, not just a type constructor, it should not be surprising to realise that once we have polymorphic mapping then we obtain polymorphic folding for free, as will be seen in the reduction rules below.

The second group of combinators are associated with the given constant functors $+$, $\times$ and 1. They are the familiar combinators for pairing, projection, inclusion, case analysis, and the canonical term of unit type.

$$\begin{aligned}
\mathrm{pair} &: \forall X_0, X_1\colon \mathrm{T}.X_0 \to X_1 \to X_0 \times X_1 \\
\mathrm{pi}_j &: \forall X_0, X_1\colon \mathrm{T}.X_0 \times X_1 \to X_j \\
\mathrm{in}_j &: \forall X_0, X_1\colon \mathrm{T}.X_j \to X_0 + X_1 \\
\mathrm{case} &: \forall X_0, X_1, Y\colon \mathrm{T}.(X_0 \to Y) \to (X_1 \to Y) \to X_0 + X_1 \to Y \\
\mathrm{un} &: 1\ .
\end{aligned}$$

3.1 Type inference

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

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

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 $RS = S'$ and $R(\tau) = \tau'$.

Milner's algorithm W (see [Mil78, Tof88]) 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.

- $W(\Delta, \Gamma, x) = (\Delta\ \Delta_1, \mathrm{id}, \tau)$, 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

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

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

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

  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.

Theorem 5. *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 [Tof88]) and use type substitution (see Lemma 4) . □

4 Term reduction and its properties

The reduction Fβ on terms of FML is defined as follows. Basic reductions (defined below) applied to a sub-term yields a one-step reduction. Then a reduction $t \to t'$ is a finite sequence of one-step reductions.

The basic reductions are given by the following rules, in which i ranges over m and j ranges over 2.

$$\begin{aligned}
(\lambda x.t_2)\ t_1 &> t_2\{t_1/x\} \\
\text{let } x = t_1 \text{ in } t_2 &> t_2\{t_1/x\} \\
\mathrm{pi}_j\ (\mathrm{pair}\ t_0\ t_1) &> t_j \\
\mathrm{case}\ f_0\ f_1\ (\mathrm{in}_j\ t) &> f_j\ t \\
\mathrm{fold}_m\ f\ (\mathrm{intro}_m\ t) &> f\ (\mathrm{map}_{m+1}\ (\lambda x.x)_{i\in m}\ (\mathrm{fold}_m\ f)\ t) \\
\mathrm{pin}_{m,i}\ (\mathrm{pex}_{m,i}\ t) &> t \\
\mathrm{din}_{m,n}\ (\mathrm{dex}_{m,n}\ t) &> t \\
\mathrm{map}_m\ f_{k\in m}\ (\mathrm{pex}_{m,i}\ t) &> \mathrm{pex}_{m,i}\ (f_i\ t) \\
\mathrm{map}_n\ f_{k\in n}\ (\mathrm{dex}_{m,n}\ t) &> \mathrm{dex}_{m,n}\ (\mathrm{map}_m\ (\mathrm{map}_n\ \overline{f})_{i\in m}\ t) \\
\mathrm{map}_m\ f_{i\in m}\ (\mathrm{intro}_m\ t) &> \mathrm{intro}_m\ (\mathrm{map}_{m+1}\ \overline{f}\ (\mathrm{map}_m\ \overline{f})\ t) \\
\mathrm{map}_2\ f_0\ f_1\ (\mathrm{pair}\ t_0\ t_1) &> \mathrm{pair}\ (f_0\ t_0)\ (f_1\ t_1) \\
\mathrm{map}_2\ f_0\ f_1\ (\mathrm{in}_j\ t) &> \mathrm{in}_j\ (f_j\ t) \\
\mathrm{map}_0\ \mathrm{un} &> \mathrm{un}
\end{aligned}$$

The reduction rules above can be classified as follows. The first two rules express β-reduction and its equivalent for let-terms. The next three rules express introduction-elimination rules for products, sums and initial algebras. The last of these may be unfamiliar. When $m = 0$ it is

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

That is $\mathrm{fold}_0\ f$ acts by recursively mapping itself across all of the substructures of $\mathrm{intro}_0\ t$ and then applying f to the result:

$$\begin{tikzcd}
F(\mu^0 F) \arrow[r, "\mathrm{intro}_0"] \arrow[d, "\mathrm{map}_1\ (\mathrm{fold}_0\ f)"'] & \mu^0 F \arrow[d, "\mathrm{fold}_0\ f"] \\
F(Y) \arrow[r, "f"'] & Y\ .
\end{tikzcd}$$

Without polymorphic mapping it would be necessary to expand this definition for each choice of functor. The next two rules reflect the status of $\mathrm{pin}_{m,i}$ etc. as isomorphisms. The remaining rules describe the action of mapping. Most interesting is the first of these, which shows how a mapping locates its data. $\mathrm{pex}_{m,i}$ identifies the datum t as being the ith argument of m so enabling the ith function argument f_i to be applied to it. Note that $\mathrm{pex}_{m,i}$ still appears in the result, since now $f_i\ t$ is the ith argument. The rules for $\mathrm{dex}_{m,n}$ and intro_m shows how to pass a mapping inside an outer functor argument. The last two rules are particular to the constant functors $+$ and $\times$ introduced to the system, and express their intended functoriality.

Theorem 6 SR. *Let $t \to t'$. If $\Delta; \Gamma \vdash t{:}\,\tau$ then $\Delta; \Gamma \vdash t'{:}\,\tau$.*

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 analyse only the last rules in the derivation of the premise, using Lemma 4 to handle term substitutions. □

Theorem 7 CR. Fβ *on untyped terms is Church-Rosser.*

Proof. Standard. The combinatory reductions rules for Fβ are left-linear and non overlapping, and one can apply the result in [Acz78] (see also [Klo80]). □

Corollary 8 CR. Fβ *on typable terms is Church-Rosser.*

Proof. Immediate from SR and CR on untyped terms. □

Theorem 9 SN. *If $\Delta; \Gamma \vdash t{:}\,\tau$, then t is strongly normalising.*

Proof. We prove SN for a system more powerful than FML, *functorial* F (briefly FF), which can type every term typable in FML (see Section 3), therefore SN for FF trivially implies SN for FML. The proof follows [Men91] and uses semantic techniques (reducibility candidates). The details are in [BJM96]. □

5 Examples

Define the polymorphic identity, and composition in the usual way, by

$$\begin{aligned} \mathrm{id} &= \lambda x.x \\ g \circ f &= \lambda x.g\ (f\ x)\ . \end{aligned}$$

Composition associates to the right. Now let us consider the list functor, $L = \mu^1 F$ where $F = +\langle 1\langle\rangle^2, \times\rangle$. Then for any type X we have the constructors

$$\begin{aligned} \mathrm{nil} &= (\mathrm{intro}_1 \circ \mathrm{dex}_{2,2} \circ \mathrm{in}_0 \circ \mathrm{dex}_{0,2})\ \mathrm{un}{:}\, LX \\ \mathrm{cons} &= \lambda x.\lambda y.(\mathrm{intro}_1 \circ \mathrm{dex}_{2,2} \circ \mathrm{in}_1)\ (\mathrm{pair}\ x\ y){:}\, X \to LX \to LX\ . \end{aligned}$$

The composite applied to un in defining nil is displayed as the top line of Figure 1. 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 $\mathrm{map}_1\ f\ \mathrm{nil}$

reduces to nil by five map-reductions, as diagrammed in Figure 1, where $g = \mathrm{map}_2\ f\ (\mathrm{map}_1\ f)$. Similarly,

$$\begin{aligned}\mathrm{map}_1\ f\ (\mathrm{cons}\ h\ t) &\to (\mathrm{intro}_1 \circ \mathrm{dex}_{2,2} \circ \mathrm{in}_1)\ (\mathrm{map}_2\ f\ (\mathrm{map}_1\ f)\ (\mathrm{pair}\ h\ t)) \\ &\to \mathrm{cons}\ (f\ h)\ (\mathrm{map}_1\ f\ t)\ .\end{aligned}$$

For folding over a list, let $d\colon D$ and $g\colon X \times D \to D$ be terms. Then we can define

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

Fig. 1. $\mathrm{map}_1 f$ nil

$$f = (\mathrm{case}\ (\lambda x.d)\ g) \circ \mathrm{din}_{2,2}\colon F(X, D) \to D\ .$$

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

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

6 Conclusions and future work

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 any functorial type constructor, 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 normalising.

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* [Jay95a] 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, e.g. those proposed in [PJ91, Ler92] to distinguish between boxed and unboxed values and providing explicit coercions between them. One can envisage an intensional semantics ([BJM96]) where $\mathrm{pex}_{m,i}(t) \in \Pi^m_i(\overline{X})$ is like a boxed value, since t is wrapped with additional information about m and i, while $\mathrm{dex}_{m,n}$ acts like data redistribution, here distinguishing between types and functors is crucial.

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

[Acz78] P. Aczel. A general Church-Rosser theorem. Technical report, Univ. of Manchester, 1978.

[Bar84] H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North Holland, 1984. revised edition.

[Bar92] H.P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*. Oxford Univ. Press, 1992.

[Ben67] J. Benabou. *Introduction to bicategories*, volume 47. Springer, 1967.

[BFSS90] E.S. Bainbridge, P.J. Freyd, A. Scedrov, and P.J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70:35–64, 1990.

[BJM96] G. Bellè, C. B. Jay, and E. Moggi. Functorial ML. available from ftp://ftp.disi.unige.it/person/MoggiE/functorial_ml.dvi, 1996.

[BW90] M. Barr and C. Wells. *Category Theory for Computing Science*. International Series in Computer Science. Prentice Hall, 1990.

[CF92] J.R.B. Cockett and T. Fukushima. About **charity**. Technical Report 92/480/18, University of Calgary, 1992.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7. CUP, 1989.

[HM95] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, January 1995.

[Jay] C.B. Jay. Type-free term reduction for covariant types. Tech. report to appear.

[Jay95a] C.B. Jay. Polynomial polymorphism. In R. Kotagiri, editor, *Proceedings of the Eighteenth Australasian Computer Science Conference: Glenelg, South*

Australia 1–3 February, 1995, volume 17, pages 237–243. A.C.S. Communications, 1995.

[Jay95b] C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.

[Jay95c] C.B. Jay. Shape analysis for parallel computing. In *Parallel Computing Workshop '95 at Fujitsu Parallel Computing Centre, Imperial College*, 1995.

[Jay96] C.B. Jay. A fresh look at parametric polymorphism: covariant types. In *Proceedings of the 19th Australasian Computer Science Conference, Melbourne, Australia, January 31–February 2 1996.*, pages 525–533, 1996.

[Jeu95] J. Jeuring. Polytypic pattern matching. In *Conference on Functional Programming Languages and Computer Architecture*, pages 238–248, 1995.

[Jon95] M.P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *J. of Functional Programming*, 5(1), 1995.

[Klo80] J.W. Klop. *Combinatory Reduction Systems*. PhD thesis, Mathematical Center Amsterdam, 1980. Tracts 129.

[Ler92] X. Leroy. Unboxed objects and polymorphic typing. In *19th Symp. on Principle of Programming Languages*. ACM Press, 1992.

[LS86] J. Lambek and P.J. Scott. *Introduction to Higher-Order Categorical Logic*, volume 7 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, 1986.

[Men91] N.P. Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51, 1991.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Procceding of the 5th ACM Conference on Functional Programming and Compter Architecture*, volume 523 of *LNCS*, pages 124–44. Springer Verlag, 1991.

[MH95] E. Meijer and G. Hutton. Bananas in space: extending fold and unfold to exponential types. In *Procedings 7th International Conference on Functional Programming and Computer Architecture, San Diego, California, June 1995.* ACM Press, 1995.

[Mil78] R. Milner. A theory of type polymorphism in programming. *JCSS*, 17, 1978.

[PJ91] S. Peyton Jones. Unboxed values as first-class citizens. In *Functional Programming and Computer Architecture*, volume 523 of *LNCS*, 1991.

[RP90] J. Reynolds and G.D. Plotkin. On functors expressible in polymorphic lambda-calculus. In G. Huet, editor, *Logical Foundations of Functional Programming*. Addison-Wesley, 1990.

[Tof88] M. Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, 1988. available as CST-52-88.

A Semantics

We consider two conventional semantics for FML types: set-theoretic (i.e. types as sets) and domain-theoretic (i.e. types as ω-cpos). Actually, to interpret type schema one should use a set theory with universes, and interpret types as small sets/cpos. The interesting part of the semantics is how to accommodate FML functors, up to minor variations we consider two interpretations: extensional (i.e. as functors) and intensional (i.e. as functor *representations*).

The main feature of intensional interpretations is that composition is no longer associative, but there is a natural isomorphism $F\langle\overline{G}\rangle^n(\overline{X}) \cong F(G_{i\in m}(\overline{X}))$, which is captured by the FML constants $\mathrm{dex}_{m,n}$ and $\mathrm{din}_{m,n}$. Similarly $\mathrm{pex}_{m,i}$ and $\mathrm{pin}_{m,i}$ capture the natural isomorphism $\Pi^m_i(\overline{X}) \cong X_i$. Intensional semantics of FML functors are important to justify some of the language design choices of FML. We expect that intentional models based on realizability (which are still under investigation) might be able to account also for the *cost* of data access and redistribution.

A.1 Set-theoretic semantics

In the category **Set** of sets and functions the obvious interpretation of an FML functor of arity m is as a functor $F\colon \mathbf{Set}^m \to \mathbf{Set}$. But the interpretation of $\mu^m F$ is problematic, since there are functors which do not have an initial algebra (for cardinality reasons), e.g. the powerset functor $P(X) = \{X' | X' \subseteq X\}$.

Shapely functors. One may avoid this problem by interpreting FML functors in a suitably restricted class of functors on **Set**. Among the many possibilities, we take shapely functors over lists (see [Jay95b]). In **Set** they are (up to natural isomorphism) those functors of the form $F(\overline{X}) = \sum_{s\in S}(\prod_{i\in m} X_i^{E(s,i)})$ for some set S and $E\colon S \to N^m$. Although adequate, this class of funtors does not include the finite powerset functor and the functor $F(X) = A + X^B$ (when B is infinite) even though they have an initial algebra.

Shapely functor representations. A more intensional semantics is to interpret FML functors of arity m as representations $\langle S \in \mathbf{Set}, E\colon S \to N^m\rangle$ of shapely functors over lists.

- Π^m_i is the representation $\langle S, E\rangle$ s.t. $S = 1 \qquad E(0, j) = \delta_{i,j}$
- $F\langle\overline{G}\rangle^n$ is the representation $\langle S, E\rangle$ s.t.

  $$S = \sum_{s\in S_F}(\prod_{i\in m} S_{G_i}^{E_F(s,i)}) \qquad E(\langle s, f\rangle, j) = \sum_{i\in m}(\sum_{e\in E_F(s,i)} E_{G_i}(f(i,e), j))$$

  where $F = \langle S_F, E_F\rangle$ and $G_i = \langle S_{G_i}, E_{G_i}\rangle$

– $\mu^m F$ is the representation $\langle S, E\rangle$ s.t.

$$S = \mu S. \sum_{s\in S_F} S^{E_F(s,m)} \qquad E(\langle s, f\rangle, i) = E_F(s,i) + \sum_{e\in E_F(s,m)} E(f\ e, i)$$

where $F = \langle S_F, E_F\rangle$. S can be viewed as the set of finite trees with nodes labeled by S_F (and branching determined by the label), while $E(\_, i)$ is a weight function (defined by induction on the structure of S) assigning weight $E_F(s,i)$ to label s.

There is an alternative representation of shapely functors over lists as formal power series $F(\overline{X}) = \sum_{n\in N^m} C(n) \times \prod_{i\in m} X_i^{n(i)}$ for some $C\colon N^m \to \mathbf{Set}$. It is not clear what are the trade-offs between the two representations. Only more accurate models, accounting also for the cost of accessing data, might help in discriminating between the two representations.

A.2 Domain-theoretic semantics

It is worth considering a domain-theoretic semantics, since it highlights some subtleties that do not arise in **Set**. In the category **Cpo** of ω-cpos and ω-continuous functions the interpretation of an FML functor of arity m as a functor $F\colon \mathbf{Cpo}^m \to \mathbf{Cpo}$ is unsatisfactory, because terms should be interpreted by continuous functions. Therefore, it is necessary to restrict to *strong* functors, where the action on morphisms is continuous. Also in **Cpo** the interpretation of $\mu^m F$ is problematic, though the difficulties can be overcome as in **Set**.

Shapely functors. In **Cpo** shapely functors over lists are (up to natural isomorphism) of the form $F(\overline{X}) = \sum_{s\in S} C(s) \times (\prod_{i\in m} X_i^{E(c,i)})$ for some set S, $C\colon S \to \mathbf{Cpo}$ and $E\colon S \to N^m$.

To interpret FML functors one may consider a more restricted class of functors, by taking $C(s) = 1$. In **Cpo** these are exactly the shapely functors over lists mapping flat cpos to flat cpos (i.e. those cpos corresponding to ML equality types).

Shapely functor representations. The intensional semantics in **Cpo** can be taken verbatim from **Set**, but now a pair $\langle S, E\colon S \to N^m\rangle$ represents a strong functor on **Cpo**, which extends the functor on **Set** with the same representation (sets can be are identified with flat cpos).

B Strong normalization

We prove SN for a system more powerful than FML, *functorial* F (briefly FF), which can type every term typable in FML (see Section 3), therefore SN for FF trivially implies SN for FML. The proof follows [Men91], which proves SN for system F extended with inductive and coinductive types using Girard's *reducibility candidates* method. More precisely, we define an interpretation $[\![-]\!]$ of types as suitable sets of strongly normalizable terms, and then prove that $\Delta;\Gamma \vdash t\colon\sigma$ implies $t \in [\![\sigma]\!]$.

B.1 System FF

System FF is obtained from FML by identifying type and type schemas, and it extends system F in the same way as FML extends ML. More precisely:

– functors are as in FML, i.e.

$$F, G ::= X \mid C \mid \Pi_i^m \mid F\langle\overline{G}\rangle^n \mid \mu^m F$$

– types and type schemas are identified (and called types), i.e.

$$\tau, \sigma ::= X \mid F(\overline{\tau}) \mid \tau \to \sigma \mid \forall X\colon \mathrm{T}.\tau \mid \forall X\colon m.\tau$$

– formation rules for type contexts $\Delta \vdash$ and functors $\Delta \vdash F\colon n$ are as in FML, while formation rules for types $\Delta \vdash \tau$ and contexts $\Delta;\Gamma \vdash$ are as in FML modulo the identification of types and type schemas.
– raw terms are as in FML, i.e.

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

– the type assignment rules for deriving $\Delta;\Gamma \vdash t\colon\tau$ in FF are

$$(x)\ \frac{\Delta;\Gamma \vdash}{\Delta;\Gamma \vdash x\colon\sigma}\ \sigma = \Gamma(x) \qquad (c)\ \frac{\Delta;\Gamma \vdash}{\Delta;\Gamma \vdash c\colon\sigma_c}$$

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

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

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

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

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

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

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

Lemma 10. *If $\Delta;\Gamma \vdash t\colon\tau$ is derivable in* FML*, then it is derivable in* FF.

Proof. The type assignment rules of FML not already in FF are (x), (c) and (let). The rule (x) of FML is derivable from (x) of FF by repeatedly applying (App) and (App$_n$). Similarly for (c). (let) of FML is derivable from (let) of FF by repeatedly applying (Λ) and (Λ_n) to the first premise. □

For the sake of simplicity, we make some changes to the set of terms:

- let $x = t_1$ in t_2 will be identified with $(\lambda x.t_2)t_1$ (this cannot be done in FML)
- combinators will be required to be *fully applied* (this is not essential for SN, but it make the definition of value much shorter shorter), as a consequence of that the type assignment rules for combinators need to be changed, e.g. the new rule for intro$_m$ becomes

$$(\mathrm{intro}_m)\ \frac{\begin{array}{c}\Delta \vdash F\colon m+1 \quad \Delta \vdash \tau_{i\in m}\\ \Delta;\Gamma \vdash t\colon F(\overline{\tau}, \mu^m F)\end{array}}{\Delta;\Gamma \vdash \mathrm{intro}_m(t)\colon \mu^m F}$$

Terms. We introduce two syntactic categories: terms $t \in \Lambda$ and values $v \in \mathrm{V}$.

$$\begin{aligned} f, t \in \Lambda &::= x \mid v \mid t_1\ t_2 \mid \mathrm{pi}_i(t) \mid \mathrm{case}(f_0, f_1, t) \mid \mathrm{fold}_m(f, t) \mid \\ &\quad\ \mathrm{pin}_{m,i}(t) \mid \mathrm{din}_{m,n}(t) \mid \mathrm{map}_m(f_{i\in m}, t) \\ v \in \mathrm{V} &::= \lambda x.t \mid \mathrm{un} \mid \mathrm{pair}(t_0, t_1) \mid \mathrm{in}_i(t) \mid \mathrm{intro}_m(t) \mid \\ &\quad\ \mathrm{pex}_{m,i}(t) \mid \mathrm{dex}_{m,n}(t) \end{aligned}$$

Lemma 11 Closure under substitution. *If S is a term substitution (i.e. a function from term variables to terms) and $v \in \mathrm{V}$, then $S(v) \in \mathrm{V}$.*

Reduction The notion of reduction Fβ on Λ is given by

- $(\lambda x.t_2)\ t_1 > t_2\{t_1/x\}$
- $\mathrm{pi}_i(\mathrm{pair}(t_0, t_1)) > t_i$
- $\mathrm{case}(f_0, f_1, \mathrm{in}_i(t)) > f_i\ t$
- $\mathrm{fold}_m(f, \mathrm{intro}_m(t)) > f\ \mathrm{map}_{m+1}((\lambda x.x)_{i\in m}, \lambda x.\mathrm{fold}_m(f, x), t)$
- $\mathrm{pin}_{m,i}(\mathrm{pex}_{m,i}(t)) > t$
- $\mathrm{din}_{m,n}(\mathrm{dex}_{m,n}(t)) > t$
- $\mathrm{map}_m(f_{j\in m}, \mathrm{pex}_{m,i}(t)) > \mathrm{pex}_{m,i}(f_i\ t)$
  $\mathrm{map}_n(f_{j\in n}, \mathrm{dex}_{m,n}(t)) > \mathrm{dex}_{m,n}(\mathrm{map}_m((\lambda x.\mathrm{map}_n(\overline{f}, x))_{i\in m}, t))$
  $\mathrm{map}_m(f_{i\in m}, \mathrm{intro}_m(t)) > \mathrm{intro}_m(\mathrm{map}_{m+1}(\overline{f}, \lambda x.\mathrm{map}_m(\overline{f}, x), t)$
  $\mathrm{map}_2(f_0, f_1, \mathrm{pair}(t_0, t_1)) > \mathrm{pair}(f_0\ t_0, f_1\ t_1)$
  $\mathrm{map}_2(f_0, f_1, \mathrm{in}_i(t)) > \mathrm{in}_i(f_i\ t)$
  $\mathrm{map}_0(\mathrm{un}) > \mathrm{un}$

We write $>$ also for the corresponding reduction, i.e. the *compatible*, reflexive and transitive closure of $>$.

B.2 Semantic domains

This section defines the sets Sat and Sat_m: Sat consists of sets for interpreting types and type schemas, while Sat_m consists of functions for interpreting functors of arity m.

Definition 12 Saturated sets.

- SN is the set of $t \in \Lambda$ which are strongly normalizable.
- $\mathrm{V}(t)$ is the set of values of t, i.e. those $v \in \mathrm{V}$ s.t. $t >^* v$.
- Sat is the set of saturated sets, i.e. those $X \subseteq \mathrm{SN}$ s.t.
  - closure under reduction: $t \in X$ and $t > t'$ imply $t' \in X$;
  - saturation: $t \in \mathrm{SN}$ and $\mathrm{V}(t) \subseteq X$ imply $t \in X$.

Lemma 13. Sat *ordered by inclusion is a complete lattice, with the least element* $\bot$ *being the set of strongly normalizable terms which cannot reduce to a value, and with meet given by intersection.*

In particular there is a least fixed-point operator lfp: $(\mathrm{Sat} \to_{mon} \mathrm{Sat}) \to_{mon} \mathrm{Sat}$, where $\mathrm{Sat} \to_{mon} \mathrm{Sat}$ is the collection of monotonic functions on Sat.

Definition 14. Given $X_1, X_2 \in \mathrm{Sat}$, the set $(X_1 \to X_2) \in \mathrm{Sat}$ is given by

$$X_1 \to X_2 = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t).\exists t_2.(v \equiv \lambda x.t_2) \wedge (\forall t_1 \in X_1.t_2\{t_1/x\} \in X_2)\}.$$

It is straightforward to prove that $(X_1 \to X_2) \in \mathrm{Sat}$.

Definition 15. A function $F\colon \mathrm{Sat}^m \to \mathrm{Sat}$ is a **functorial operator** iff for any $X_{i\in m}, Y_{i\in m} \in \mathrm{Sat}$ the following properties hold

- monotonicity: $(\forall i \in m.X_i \subseteq Y_i)$ implies $F(\overline{X}) \subseteq F(\overline{Y})$;
- functoriality: $(\forall i \in m.f_i \in (X_i \to Y_i))$ and $t \in F(\overline{X})$ imply $\mathrm{map}_m(\overline{f}, t) \in F(\overline{Y})$.

Sat_m is the set of functorial operators of arity m.

B.3 Semantic functions

This section introduce the semantic counterpart of functor constructors, i.e.

- $[\![1]\!] \in \mathrm{Sat}_0$ and $[\![\times]\!], [\![+]\!] \in \mathrm{Sat}_2$ interpret the functor constants;
- $\mathbf{P}^m_i \in \mathrm{Sat}_m$ interprets the projection Π^m_i;
- $\mathbf{C}_{m,n}\colon \mathrm{Sat}_m \to \mathrm{Sat}^m_n \to \mathrm{Sat}_n$ interprets functor composition $F\langle G_{i\in m}\rangle^n$;
- $\mathbf{m}_m\colon \mathrm{Sat}_{m+1} \to \mathrm{Sat}_m$ interprets the inductive functor construction $\mu^m F$.

The semantic counterpart of type constructors is *as expected* (see [Men91]), i.e.

- functional types are interpreted by saturated sets of the form $X_1 \to X_2$;
- universal quantification is interpreted by intersection (on Sat or Sat_m);
- functor application is interpreted by function application.

After the definition of the semantics function, we sketch a proof that they have the expected codomain (this is tricky only for $\mathbf{m}_m$, where the proof mimics that for inductive types given in [Men91]).

Definition 16.

1. $[\![1]\!]() = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t). v \equiv \mathrm{un}\}$.
2. $[\![\times]\!](X_0, X_1) = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t). \exists t_0 \in X_0. \exists t_1 \in X_1. v \equiv \mathrm{pair}(t_0, t_1)\}$.
3. $[\![+]\!](X_0, X_1) = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t). \exists i \in 2. \exists t' \in X_i. v \equiv \mathrm{in}_i(t')\}$.
4. $\mathbf{P}^m_i(\overline{X}) = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t). \exists t' \in X_i. v \equiv \mathrm{pex}_{m,i}(t')\}$.
5. $\mathbf{C}_{m,n}(F, \overline{G})(\overline{X}) = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t). \exists t' \in F(G_i(\overline{X})_{i \in m}). v \equiv \mathrm{dex}_{m,n}(t')\}$.
6. $\mathbf{m}_m(F)(\overline{X}) = \mathrm{lfp}\, H_{F,\overline{X}}$, where $H_{F,\overline{X}} \colon \mathrm{Sat} \to_{mon} \mathrm{Sat}$ is
   $H_{F,\overline{X}}(X) = \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t). \exists t' \in F(\overline{X}, X). v \equiv \mathrm{intro}_m(t')\}$.

Lemma 17. *The semantics functions in Definition 16 are well defined, i.e.*

1. $[\![1]\!] \in \mathrm{Sat}_0$.
2. $[\![\times]\!] \in \mathrm{Sat}_2$.
3. $[\![+]\!] \in \mathrm{Sat}_2$.
4. $\mathbf{P}^m_i \in \mathrm{Sat}_m$.
5. *If* $F \in \mathrm{Sat}_m$ *and* $G_{i \in m} \in \mathrm{Sat}_n$, *then* $\mathbf{C}_{m,n}(F, \overline{G}) \in \mathrm{Sat}_n$.
6. *If* $F \in \mathrm{Sat}_{m+1}$, *then* $\mathbf{m}_m(F) \in \mathrm{Sat}_m$.

Proof.

1. We have to prove that $[\![1]\!] \in \mathrm{Sat}_0$.
   - $[\![1]\!]() \in \mathrm{Sat}$. Closure under reduction: if $t \in [\![1]\!]()$, $t > t'$, and $v \in \mathrm{V}(t')$, then $v \in \mathrm{V}(t)$, and so $v \equiv \mathrm{un}$. Saturation: let $t \in \mathrm{SN}$ and $\mathrm{V}(t) \subseteq [\![1]\!]()$, then for all $v \in \mathrm{V}(t)$ we have $v \equiv \mathrm{un}$, hence $t \in [\![1]\!]()$.
   - Monotonicity is trivial since the arity of the functor is 0.
   - Functoriality:
     let $t \in [\![1]\!]()$. First we have to show that $\mathrm{map}_0(t) \in \mathrm{SN}$. Suppose it has an infinite reduction path. Then because t is strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when t has been reduced to un, and then the reduction yields un; contradiction. This shows also that all values of $\mathrm{map}_0(t)$ are of the form un and hence that $\mathrm{map}_0(t) \in [\![1]\!]()$.
2. We have to prove that $[\![\times]\!] \in \mathrm{Sat}_2$.
   - $[\![\times]\!](X_0, X_1) \in \mathrm{Sat}$. Closure under reduction: it follows from the fact that $\mathrm{V}(t') \subseteq \mathrm{V}(t)$ whenever $t > t'$: let $t \in [\![\times]\!](X_0, X_1)$, if $t > t'$ then t' is strongly normalizable and for all $v \in \mathrm{V}(t')$, there exist $t_0 \in X_0, t_1 \in X_1$, such that $v \equiv \mathrm{pair}(t_0, t_1)$. Saturation: let $t \in \mathrm{SN}$, if $\mathrm{V}(t) \subseteq [\![\times]\!](X_0, X_1)$ then for all $v \in \mathrm{V}(t)$, there exist $t_0 \in X_0, t_1 \in X_1$, such that $v \equiv \mathrm{pair}(t_0, t_1)$ and this is the required.
   - Monotonicity: straightforward.

  - Functoriality:
    let $f_i \in (X_i \to Y_i)$, for $i \in 2$ and let $t \in [\![\times]\!](X_0, X_1)$. We are to show that $\mathrm{map}_2(f_0, f_1, t) \in [\![\times]\!](Y_0, Y_1)$. First we will show that $\mathrm{map}_2(f_0, f_1, t) \in \mathrm{SN}$. Suppose by absurd that there exists an infinite reduction path. Then because f_0, f_1 and t are strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when t has been reduced to a value. Since $t \in [\![\times]\!](X_0, X_1)$ then the value is of the form $\mathrm{pair}(t_0, t_1)$. Therefore the outermost step yields $\mathrm{pair}(f_0' t_0, f_1' t_1)$, where f_0', f_1' are reducts respectively of f_0, f_1, which are strongly normalizable since f_i' is in $(X_i \to Y_i)$. Now we show that if $v \in \mathrm{V}(\mathrm{map}_2(f_0, f_1, t))$ then there are terms $t_0 \in Y_0, t_1 \in Y_1$ such that $v \equiv \mathrm{pair}(t_0, t_1)$. Consider such a v. It has been obtained from $\mathrm{map}_2(f_0, f_1, t)$ by a sequence of non-outermost reduction steps and, once t has been reduced to a value, by an outermost reduction step. But t can only reduce to a value of the form $\mathrm{pair}(t_0, t_1)$ since t is in $[\![\times]\!](X_0, X_1)$. So v has the form $\mathrm{pair}(t_0', t_1')$ for some t_i' that are reducts of $f_i' t_i$. Since $f_i \in (X_i \to Y_i)$ it follows that $f_i' t_i \in Y_i$ and, by closure under reduction, we have that $t_i' \in Y_i$ as required.

3. Proceed as in the case for $[\![\times]\!]$.
4. Proceed as in the case for $[\![\times]\!]$.
5. We have to prove that $\mathbf{C}_{m,n}(F, \overline{G}) \in \mathrm{Sat}_n$.
  - $\mathbf{C}_{m,n}(F, \overline{G})(\overline{X}) \in \mathrm{Sat}$. Closure under reduction and saturation: proceed as in the case for $[\![\times]\!]$.
  - Monotonicity: straightforward.
  - Functoriality:
    let $Y_{j \in n}$, $f_j \in (X_j \to Y_j)$, for $j \in n$ and let $t \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{X})$. We are to show that $\mathrm{map}_n(\overline{f}, t) \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{Y})$. First we show that it is strongly normalizable. Suppose by absurd that there exists an infinite reduction path. Then because $f_{j \in n}$ and t are strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when t has been reduced to value. Since $t \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{X})$ then t has been reduced to a value of the form $\mathrm{dex}_{m,n}(t_1)$. Therefore the outermost step yields $(\mathrm{dex}_{m,n}(\mathrm{map}_m((\lambda x.\mathrm{map}_n(\overline{f}, x))_{i \in m}, t_1))$.
    Now suppose that $\lambda x.\mathrm{map}_n(\overline{f}, x) \in G_i(\overline{X}) \to G_i(\overline{Y})$ for all $i \in m$. Clearly $t_1 \in F(G_i(\overline{X})_{i \in m})$. Then we have that $\mathrm{map}_m((\lambda x.\mathrm{map}_n(\overline{f}, x))_{i \in m}, t_1) \in F(G_i(\overline{Y})_{i \in m})$ by functoriality of F, which gives the desired contradiction.
    We are left with showing that $(\lambda x.\mathrm{map}_n(\overline{f}, x))$ is in $G_i(\overline{X}) \to G_i(\overline{Y})$ for all $i \in m$. Let i be arbitrary. Then $\lambda x.\mathrm{map}_n(\overline{f}, x)$ is in SN since reduction can only occur inside the subterm f which by hypothesis is strongly normalizable. Consider a value v of $\lambda x.\mathrm{map}_n(\overline{f}, x)$. This v is of the form $\lambda x.\mathrm{map}_n(\overline{f'}, x)$, where each f_j' is a reduct of the corresponding f_j. By the closure condition we have that $f_j' \in X_j \to Y_j$. By functoriality of G_i, for all $t \in G_i(\overline{X})$ we have that $\mathrm{map}_n(\overline{f'}, t) \in G_i(\overline{Y})$. Now we

show that if $v \in \mathrm{V}(\mathrm{map}_n(\overline{f}, t))$ then there is a term $t_1 \in F(G_i(\overline{X})_{i\in m})$ such that $v \equiv \mathrm{dex}_{m,n}(t_1)$. Consider such a v. It has been obtained from $\mathrm{map}_n(\overline{f}, t)$ by a sequence of non-outermost reduction steps and, once that t has been reduced to a value, by an outermost reduction step. But t can only reduce to a value of the form $\mathrm{dex}_{m,n}(t_1)$ since $t \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{X})$. So v has the form $\mathrm{dex}_{m,n}(t_2)$ for some t_2 that are reducts of $(\mathrm{map}_m((\lambda x.\mathrm{map}_n(\overline{f}, x))_{i\in m}, t_1)$. Since $\lambda x.\mathrm{map}_n(\overline{f}, x) \in G_i(\overline{X}) \to G_i(\overline{Y})$ for all $i \in m$ and $t_1 \in F(G_i(\overline{X})_{i\in m})$, by functoriality of F we can conclude that $(\mathrm{map}_m((\lambda x.\mathrm{map}_n(\overline{f}, x))_{i\in m}, t_1) \in F(G_i(\overline{Y})_{i\in m})$. By closure under reduction we have $t_2 \in F(G_i(\overline{Y})_{i\in m})$.

6. We have to prove that $\mathbf{m}_m(F) \in \mathrm{Sat}_m$. We show that:
   (a) $H_{F,\overline{X}}$ is well-defined, i.e. $\forall X \in \mathrm{Sat}.H_{F,\overline{X}}(X) \in \mathrm{Sat}$.
   (b) $H_{F,\overline{X}}\colon \mathrm{Sat} \to_{mon} \mathrm{Sat}$ is monotone; hence there exists the least fixed-point.
   (c) $\mathbf{m}_m(F)$ satisfies monotonicity and functoriality.

   (a) We proceed as in the case for $[\![\times]\!]$.
   (b) Straightforward.
   (c) – Monotonicity:
   follows from monotonicity of the least fixed-point operator.
   – Functoriality:
   let $\overline{X}$ and $\overline{Y}$ be fixed. We have to prove that for all $i \in m$, $f_i \in (X_i \to Y_i)$ and that $t \in \mathbf{m}_m(F)(\overline{X})$ imply $\mathrm{map}_m(\overline{f}, t) \in \mathbf{m}_m(F)(\overline{Y})$. The chain for the least fixed-point $\mathbf{m}_m(F)(\overline{X})$ is

$$\begin{aligned}
\xi_0 &= \bot \\
\xi_{\alpha+1} &= \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t).\exists t' \in F(\overline{X}, \xi_\alpha).v \equiv \mathrm{intro}_m(t')\} \\
\xi_\lambda &= \textstyle\bigcup_{\alpha<\lambda} \xi_\alpha \quad \text{for limit } \lambda.
\end{aligned}$$

   The chain for the least fixed-point $\mathbf{m}_m(F)(\overline{Y})$ is

$$\begin{aligned}
\eta_0 &= \bot \\
\eta_{\alpha+1} &= \{t \in \mathrm{SN} | \forall v \in \mathrm{V}(t).\exists t' \in F(\overline{X}, \eta_\alpha).v \equiv \mathrm{intro}_m(t')\} \\
\eta_\lambda &= \textstyle\bigcup_{\alpha<\lambda} \eta_\alpha \quad \text{for limit } \lambda.
\end{aligned}$$

   By ordinal induction we prove that fixed X_i and Y_i, for all α we have that $\forall (f_i \in X_i \to Y_i)_{i\in m}.\forall t \in \xi_\alpha.\mathrm{map}_m(\overline{f}, t) \in \eta_\alpha$.
   Basis: Let $t \in \xi_0$. Since t cannot reduce to any value, neither can the term $\mathrm{map}_m(\overline{f}, t)$. Hence $\mathrm{map}_m(\overline{f}, t) \in \bot = \eta_0$.
   Inductive Case: Let $t \in \xi_{\alpha+1}$. We have to show that $\mathrm{map}_m(\overline{f}, t) \in \eta_{\alpha+1}$:
   - $\mathrm{map}_m(\overline{f}, t) \in \mathrm{SN}$. Suppose by absurd that there exists an infinite reduction path. Then because $f_{i\in m}$ and t are strongly normalizable at least one outermost reduction has been performed. By inspection of the reduction rules, an outermost reduction can only be performed when t has been reduced to a value. Since $t \in \xi_{\alpha+1}$ then the value is of the form $\mathrm{intro}_m(t_1)$. Therefore the outermost step yields $(\mathrm{intro}_m(\mathrm{map}_{m+1}(\overline{f'}, \lambda x.\mathrm{map}_m(\overline{f'}, x), t_1)))$,

where each f_i' is a reduct of the corresponding f_i. We can conclude provided we show that $(\text{map}_{m+1}(\overline{f'}, \lambda x.\text{map}_m(\overline{f'}, x), t_1)) \in F(\overline{Y}, \eta_\alpha)$. Consider $\lambda x.\text{map}_m(\overline{f'}, x)$. This term belongs to $\xi_\alpha \to \eta_\alpha$:

∗ $\lambda x.\text{map}_m(\overline{f'}, x) \in \text{SN}$: follows from the fact that a reduction can only occur inside the subterms f_i' which by hypothesis are strongly normalizable.

∗ Consider a value v of $\lambda x.\text{map}_m(\overline{f'}, x)$. This v is of the form $\lambda x.\text{map}_m(\overline{f''}, x)$, where each f_i'' is a reduct of the corresponding f_i'. By the closure condition we have that $f_i'' \in X_i \to Y_i$. By induction we get that $\forall t \in \xi_\alpha.\text{map}_m(\overline{f''}, t) \in \eta_\alpha$.

Since $f_i' \in (X_i \to Y_i)$, for all $i \in m$, $\lambda x.\text{map}_m(\overline{f'}, x) \in \xi_\alpha \to \eta_\alpha$ and $t_1 \in F(\overline{X}, \xi_\alpha)$, we have $(\text{map}_{m+1}(\overline{f'}, \lambda x.\text{map}_m(\overline{f'}, x), t_1)) \in F(\overline{Y}, \eta_\alpha)$ by functoriality of F

- Now we show that if $v \in \text{V}(\text{map}_m(\overline{f}, t))$ then there is a term $t' \in F(\overline{X}, \eta_\alpha)$ such that $v \equiv \text{intro}_m(t')$.
  Consider such a v. It has been obtained from $\text{map}_m(\overline{f}, t)$ by a sequence of non-outermost reduction steps and once t has been reduced to a value, by an outermost reduction step. But t can only reduce to a value of the form $\text{intro}_m(t_1)$. So v has the form $\text{intro}_m(t_2)$ where t_2 is a reduct of $(\text{map}_{m+1}(\overline{f'}, \lambda x.\text{map}_m(\overline{f'}, x), t_1))$. Since $(\text{map}_{m+1}(\overline{f'}, \lambda x.\text{map}_m(\overline{f'}, x), t_1)) \in F(\overline{Y}, \eta_\alpha)$, by closure under reduction we can conclude.

Limit Case: Let $t \in \xi_\lambda$. Then $t \in \xi_\alpha$ for some $\alpha \in \lambda$. By inductive hypothesis $(\text{map}_m(\overline{f}, t)) \in \eta_\alpha$. Hence $(\text{map}_m(\overline{f}, t)) \in \eta_\alpha \subseteq \eta_\lambda$.

□

B.4 Interpretation

Using the semantic functions we define an interpretation of well-formed type contexts, functors and types by induction on the derivation of $\Delta \vdash J$.

– If $\Delta \vdash$, then $[\![\Delta]\!]$ is a set

$$[\![\emptyset]\!] = 1$$
$$[\![\Delta, X\colon n]\!] = [\![\Delta]\!] \times \text{Sat}_n$$
$$[\![\Delta, X\colon \text{T}]\!] = [\![\Delta]\!] \times \text{Sat}$$

– If $\Delta \vdash F\colon n$, then $[\![\Delta.F]\!]\colon [\![\Delta]\!] \to \text{Sat}_n$

$$[\![\Delta.X]\!]\rho = \rho_X$$
$$[\![\Delta.C]\!]\rho = [\![C]\!]$$
$$[\![\Delta.\Pi_i^m]\!]\rho = \mathbf{P}_i^m$$
$$[\![\Delta.F\langle\overline{G}\rangle^n]\!]\rho = \mathbf{C}_{m,n}(H, \overline{K})$$
$$[\![\Delta.\mu^m F']\!]\rho = \mathbf{m}_m(H')$$

where $H = [\![\Delta.F]\!]\rho \in \mathrm{Sat}_m$, $H' = [\![\Delta.F']\!]\rho \in \mathrm{Sat}_{m+1}$ and $K_i = [\![\Delta.G_i]\!]\rho \in \mathrm{Sat}_n$.

– If $\Delta \vdash \tau$, then $[\![\Delta.\tau]\!]\colon [\![\Delta]\!] \to \mathrm{Sat}$

$$\begin{aligned}
[\![\Delta.X]\!]\rho &= \rho_X \\
[\![\Delta.F(\overline{\tau})]\!]\rho &= H(\overline{Y}) \\
[\![\Delta.\tau_1 \to \tau_2]\!]\rho &= Y_1 \to Y_2 \\
[\![\Delta.(\forall X\colon m.\tau)]\!]\rho &= \bigcap_{H \in \mathrm{Sat}_m} [\![\Delta\ X\colon m.\tau]\!]\langle \rho, H\rangle \\
[\![\Delta.(\forall X\colon \mathrm{T}.\tau)]\!]\rho &= \bigcap_{Z \in \mathrm{Sat}} [\![\Delta\ X\colon \mathrm{T}.\tau]\!]\langle \rho, Z\rangle
\end{aligned}$$

where $H = [\![\Delta.F]\!]\rho \in \mathrm{Sat}_m$, $Y_i = [\![\Delta.\tau_i]\!]\rho \in \mathrm{Sat}$.

– If $\Delta; \Gamma \vdash$, then $[\![\Delta.\Gamma]\!]\colon [\![\Delta]\!] \to \mathrm{DV}(\Gamma) \to \mathrm{Sat}$

$$[\![\Delta.\Gamma]\!]\rho\ x = [\![\Delta.\tau]\!]\rho \quad \text{where } \tau = \Gamma(x).$$

B.5 Strong normalization proof

The following lemma shows that the terms are well behaved with respect to the semantic functions. We will need it to prove the Proposition 19.

Lemma 18. *Let $X, Y, X_{i\in m}, Y_{i\in m}, Z_{j\in n} \in \mathrm{Sat}$, $F \in \mathrm{Sat}_m$, $G_{i\in m} \in \mathrm{Sat}_n$ and $K \in \mathrm{Sat}_{m+1}$. The following hold:*

1. $\mathrm{un} \in [\![1]\!]()$;
2. *if* $\forall t_1 \in X_1.t\{t_1/x\} \in X_2$ *then* $\lambda x.t \in (X_1 \to X_2)$;
3. *if* $t \in X_1 \to X_2$ *and* $t_1 \in X_1$ *then* $tt_1 \in X_2$;
4. *if for all* $i \in 2$, $t_i \in X_i$, *then* $\mathrm{pair}(t_0, t_1) \in [\![\times]\!](X_0, X_1)$;
5. *if* $t \in [\![\times]\!](X_0, X_1)$ *then for all* $i \in 2$, $\mathrm{pi}_i(t) \in X_i$;
6. *for all* $i \in 2$, *if* $t \in X_i$ *then* $\mathrm{in}_i(t) \in [\![+]\!](X_0, X_1)$;
7. *if for all* $i \in 2$,$f_i \in X_i \to Y$ *and* $t \in [\![+]\!](X_0, X_1)$ *then* $\mathrm{case}(f_0, f_1, t) \in Y$;
8. *for all* $i \in m$, *if* $t \in X_i$ *then* $\mathrm{pex}_{m,i}(t) \in \mathbf{P}^m_i(\overline{X})$;
9. *for all* $i \in m$, *if* $t \in \mathbf{P}^m_i(\overline{X})$ *then* $\mathrm{pin}_{m,i}(t) \in X_i$;
10. *if* $t \in F(G_i(\overline{Z})_{i\in m})$ *then* $\mathrm{dex}_{m,n}(t) \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{Z})$;
11. *if* $t \in \mathbf{C}_{m,n}(F, \overline{G})(\overline{Z})$ *then* $\mathrm{din}_{m,n}(t) \in F(G_i(\overline{Z})_{i\in m})$;
12. *if* $t \in K(\overline{X}, \mathbf{m}_m(K)(\overline{X}))$ *then* $\mathrm{intro}_m(t) \in \mathbf{m}_m(K)(\overline{X})$;
13. *if* $f \in (K(\overline{X}, Y) \to Y)$ *and* $t \in \mathbf{m}_m(K)(\overline{X})$ *then* $\mathrm{fold}_m(f, t) \in Y$;
14. *if for all* $i \in m$,$f_i \in (X_i \to Y_i)$ *and* $t \in F(\overline{X})$ *then* $\mathrm{map}_m(\overline{f}, t) \in F(\overline{Y})$.

Proof. The proof is given only for the last three cases (for the others is similar).

– Given $K \in \mathrm{Sat}_{m+1}$, $X_{i\in m} \in \mathrm{Sat}$ and $Y \in \mathrm{Sat}$, if $t \in K(\overline{X}, \mathbf{m}_m(K)(\overline{X}))$ then $\mathrm{intro}_m(t) \in \mathbf{m}_m(K)(\overline{X})$:
Since $\mathbf{m}_m(K)(\overline{X}) = H_{K,\overline{X}}(\mathbf{m}_m(K)(\overline{X}))$, where $H_{K,\overline{X}}$ is the monotonic map in the definition of $\mathbf{m}_m(K)$, we prove that $\mathrm{intro}_m(t) \in H_{K,\overline{X}}(\mathbf{m}_m(K)(\overline{X}))$.

We have that $\mathrm{intro}_m(t) \in \mathrm{SN}$ since $t \in \mathrm{SN}$. Now, a value of $\mathrm{intro}_m(t)$ is of the form $\mathrm{intro}_m(t')$, where t' is a reduct of t. Since $t' \in K(\overline{X}, \mathbf{m}_m(K)(\overline{X}))$ we have that $\mathrm{intro}_m(t) \in H_{K,\overline{X}}(\mathbf{m}_m(K)(\overline{X})) = \mathbf{m}_m(K)(\overline{X})$.

- Given $K \in \mathrm{Sat}_{m+1}$, $X_{i\in m} \in \mathrm{Sat}$ and $Y \in \mathrm{Sat}$, if $f \in (K(\overline{X}, Y) \to Y)$ and $t \in \mathbf{m}_m(K)(\overline{X})$, then $\mathrm{fold}_m(f,t) \in Y$:
  Let $(\xi_\alpha)_{\alpha\in Ord}$ be the chain for the least fixed-point $\mathbf{m}_m(K)(\overline{X})$, we prove by induction on α that

  $$\forall f \in (K(\overline{X}, Y) \to Y).\forall t \in \xi_\alpha.\mathrm{fold}_m(f,t) \in Y$$

  Basis: Let f be a term in $K(\overline{X}, Y) \to Y$. If $t \in \xi_0$ then $\mathrm{fold}_m(f,t)$ cannot reduce to a value so it belongs to $\bot \subseteq Y$.
  Inductive Case: Suppose that $t \in \xi_{\alpha+1}$. Let $f \in (K(\overline{X}, Y) \to Y)$. If t does not reduce to a value then neither $\mathrm{fold}_m(f,t)$ cannot reduce to a value, so $\mathrm{fold}_m(f,t)$ belongs to $\bot \subseteq Y$. Otherwise, t reduces to a value of the form $\mathrm{intro}_m(t_1)$, where $t_1 \in K(\overline{X}, \xi_\alpha)$. Hence $\mathrm{fold}_m(f,t)$ reduces to $f'(\mathrm{map}_{m+1}((\lambda x.x)_{i\in m}, \lambda x.\mathrm{fold}_m(f',x), t_1))$ where f' is a reduct of f. Using inductive hypothesis we get $\lambda x.\mathrm{fold}_m(f',x) \in (\xi_\alpha \to Y)$. Functoriality of K allows to conclude that $(\mathrm{map}_{m+1}((\lambda x.x)_{i\in m}, \lambda x.\mathrm{fold}_m(f',x), t_1)) \in K(\overline{X}, Y)$. Since $f \in (K(\overline{X}, Y) \to Y)$ by hypothesis, we have $\mathrm{fold}_m(f,t) \in Y$.
  Limit Case: Suppose that $t \in \xi_\lambda$. Then $t \in \xi_\alpha$ for some $\alpha \in \lambda$. By inductive hypothesis, we have $\mathrm{fold}_m(f,t) \in Y$, since $f \in (K(\overline{X}, Y) \to Y)$.
- Given $F \in \mathrm{Sat}_m$ and $X_{i\in m}, Y_{i\in m} \in \mathrm{Sat}$, if $f_i \in (X_i \to Y_i)$ and $t \in F(\overline{X})$, then $\mathrm{map}_m(\overline{f}, t) \in F(\overline{Y})$:
  Immediate from the definition of $F \in \mathrm{Sat}_m$.

□

Proposition 19. *If $\Delta; \Gamma \vdash t : \tau$ is derivable in* FF*, then $S(t) \in [\![\Delta.\tau]\!]\rho$ for any $\rho \in [\![\Delta]\!]$ and term substitution $S \in [\![\Delta.\Gamma]\!]\rho$.*

Proof. The proof is by induction on the derivation of the judgement $\Delta; \Gamma \vdash t : \tau$. Here are the justifications for the rules intro_m, fold_m and map_m.

- $(\mathrm{intro}_m)\ \dfrac{\begin{array}{c}\Delta \vdash F : m+1 \quad \Delta \vdash \tau_{i\in m}\\ \Delta; \Gamma \vdash t : F(\overline{\tau}, \mu^m F(\overline{\tau}))\end{array}}{\Delta; \Gamma \vdash \mathrm{intro}_m(t) : \mu^m F(\overline{\tau})}$
  By inductive hypothesis

  $$S(t) \in [\![\Delta.F(\overline{\tau}, \mu^m F(\overline{\tau}))]\!]\rho = K(\overline{X}, \mathbf{m}_m(K)(\overline{X}))$$

  where $K = [\![\Delta.F]\!]\rho$ and $X_i = [\![\Delta.\tau_i]\!]\rho$. By Lemma 18 we can conclude

  $$S(\mathrm{intro}_m(t)) \equiv \mathrm{intro}_m(S(t)) \in \mathbf{m}_m(K)(\overline{X}) = [\![\Delta.\mu^m F(\overline{\tau})]\!]\rho$$

- $(\mathrm{fold}_m)\ \dfrac{\begin{array}{c}\Delta \vdash F : m+1 \quad \Delta \vdash \tau_{i\in m}, \sigma\\ \Delta; \Gamma \vdash f : F(\overline{\tau}, \sigma) \to \sigma \quad \Delta; \Gamma \vdash t : \mu^m F(\overline{\tau})\end{array}}{\Delta; \Gamma \vdash \mathrm{fold}_m(f,t) : \sigma}$

By inductive hypothesis $S(f) \in K(\overline{X}, Y) \to Y$ and $S(t) \in \mathbf{m}_m(K)(\overline{X})$, where $K = [\![\Delta.F]\!]\rho$, $X_i = [\![\Delta.\tau_i]\!]\rho$ and $Y = [\![\Delta.\sigma]\!]\rho$.
By Lemma 18 we can conclude

$$S(\text{fold}_m(f,t)) \equiv \text{fold}_m(S(f), S(t)) \in Y = [\![\Delta.\sigma]\!]\rho$$

– (map_m) $\dfrac{\begin{array}{c}\Delta \vdash F{:}\, m \quad \Delta \vdash \tau_{i\in m}, \sigma_{i\in m} \\ \Delta; \Gamma \vdash f_i{:}\, \tau_i \to \sigma_i \quad \Delta; \Gamma \vdash t{:}\, F(\overline{\tau})\end{array}}{\Delta; \Gamma \vdash \text{map}_m(\overline{f}, t){:}\, F(\overline{\sigma})}$

By inductive hypothesis $S(f_i) \in X_i \to Y_i$ (for $i \in m$) and $S(t) \in H(\overline{X})$, where $H = [\![\Delta.F]\!]\rho$, $X_i = [\![\Delta.\tau_i]\!]\rho$ and $Y = [\![\Delta.\sigma]\!]\rho$.
by Lemma 18 we can conclude

$$S(\text{map}_m(\overline{f}, t)) \equiv \text{map}_m((S(f_i))_{i\in m}, S(t)) \in H(\overline{Y}) = [\![\Delta.F(\overline{\sigma})]\!]$$

□

Theorem 20 SN. *If $\Delta; \Gamma \vdash t{:}\, \tau$ is derivable in* FF*, then* $t \in \text{SN}$.

Proof. From $\Delta; \Gamma \vdash t{:}\, \tau$, by Proposition 19, we have $S(t) \in [\![\Delta.\tau]\!]\rho$, whenever $\rho \in [\![\Delta]\!]$ and $S \in [\![\Delta.\Gamma]\!]\rho$. Every $X \in \text{Sat}$ contains the term variables because of saturation: in fact $x \in \text{SN}$ and $\text{V}(x)$ is empty. Therefore, by taking S to be the identity substitution, we have that $t \in [\![\Delta.\tau]\!]\rho$ and so $t \in \text{SN}$. □

Table of Contents