Reasoning about modular datatypes with Mendler induction

Paolo Torrini Tom Schrijvers

Department of Computer Science, KU Leuven, Belgium
{p.torrini, tom.schrijvers}@cs.kuleuven.be

In functional programming, datatypes à la carte provide a convenient modular representation of recursive datatypes, based on their initial algebra semantics. Unfortunately it is highly challenging to implement this technique in proof assistants that are based on type theory, like Coq. The reason is that it involves type definitions, such as those of type-level fixpoint operators, that are not strictly positive. The known work-around of impredicative encodings is problematic, insofar as it impedes conventional inductive reasoning. Weak induction principles can be used instead, but they considerably complicate proofs.

This paper proposes a novel and simpler technique to reason inductively about impredicative encodings, based on Mendler-style induction. This technique involves dispensing with dependent induction, ensuring that datatypes can be lifted to predicates and relying on relational formulations. A case study on proving subject reduction for structural operational semantics illustrates that the approach enables modular proofs, and that these proofs are essentially similar to conventional ones.

1 Introduction

Developing high-quality software artifacts, including programs as well as programming languages, can be very expensive, and so can formally proving their properties. This makes it highly desirable to maximise reuse and extensibility. Modularity plays an essential role in this context: a component is modular whenever it can be specified independently of the whole collection – therefore, a modular characterisation of an artifact implies that its extension does not require changes to what is already in stock.

In functional programming, it is natural to rely on a structured characterisation of components based on recursive datatypes. However, conventional datatypes are not extensible – each one fixes a closed set of constructors with respect to which case analysis may have to be exhaustive, hence each case implicitly depends on the whole collection. An elegant solution to this tension between structural characterisation and modularity, also known as the *expression problem*, has been found with the notion of *modular datatype* (MDT) – i.e., datatypes *à la carte*, introduced in Haskell by Swierstra [16]. The definition of an MDT consists of two distinct parts: the grammar, as a non-recursive structure based on a functor, and the recursive datatype, as the recursive closure of the functor by a type-level fixed point. Grammar functors behave as modules, as they can be defined independently and combined together by coproduct.

In Haskell, an MDT can be easily implemented in terms of conventional datatypes, which can be used to define the grammar as well as the recursive closure (as recalled in Section 2). However, Haskell's datatype definition of the type-level fixpoint operator is not strictly positive, and therefore it is problematic from the point of view of less liberal type systems. As a general-purpose programming language, Haskell relies on types that do not enforce totality (i.e., either termination or productivity). This makes type checking easier in the presence of non-termination. Unfortunately, allowing for non-total programs can lead to inconsistency under a program-as-proof interpretation. For this reason, proof assistants based on the Curry-Howard correspondence are usually based on more restrictive type systems. Proof assistants such as Coq, Agda, Isabelle and Twelf, for instance, rely on a syntactic criterion of monotonicity

R. Matthes, M. Mio (Eds.): Fixed Points
in Computer Science 2015 (FICS 2015)
EPTCS 191, 2015, pp. 143–157, doi:10.4204/EPTCS.191.13

which ensures totality, by requiring that all the occurrences of an inductive datatype in its definition are strictly positive – hence incompatibly with the Haskell-style representation of MDTs.

Coq is a theorem prover based on the calculus of inductive constructions (CIC) [3] which extends the calculus of constructions (CC) [5] with inductive and coinductive definitions. CC, the most expressive system of the lambda cube [2], allows for types depending on terms, type-level functions and full parametric polymorphism, hence also for definitions that are impredicative, in the sense of referring in their bodies to collections that are being defined. One of the main approaches to represent MDT in Coq, due to Delaware, Oliveira and Schrijvers [7] and implemented in the MTC/3MT framework [6], takes advantage of impredicativity, and relies on the Church encoding of fixed points (as recalled in Section 3). Another promising approach, due to Keuchel and Schrijvers [11], relies on containers – it is predicative, but it involves a more indirect representation of types. Church encodings are purely based on CC and do not involve any extra-logical machinery – however, they rather complicate inductive reasoning. Impredicative definitions have an eliminative character that hides term structure, hence making it harder to reason by induction. The solution proposed by Delaware *et al.* is quite general – however, it relies on proof algebras that pack terms together with proofs using Σ-types, and this leads to inductive proofs that have a significant overhead with respect to the conventional, non-modular ones.

This paper proposes a novel solution to the problem of reasoning inductively with impredicatively encoded MDT, based on the use of Mendler-style induction [12, 18, 1]. Mendler's characterisation of iteration makes it possible to encode an induction principle within the impredicative encoding of an MDT. Unlike Delaware *et al.*, we use Mendler algebras as proof algebras. This leads to inductive proofs that are straightforwardly modular and ultimately closer to conventional ones (Section 4). Although this approach cannot handle dependent induction, this limitation is of little consequence as long as we are reasoning about relational formulations. Nonetheless, this may make it necessary to lift inductive datatypes to inductively defined predicates, in order to use them as inductive arguments in proofs.

In order to reason inductively on relations, we clearly need to rely on functor shapes that can represent them as well as mutual dependencies. Such need is highlighted throughout a case study on the formalisation of a language based on structural operational semantics (Section 5, Coq implementation available [17]). The language, for which we prove type preservation, has a definition that involves mutual dependency between expressions and declarations.

2 Datatypes a-la-carte

MDTs as introduced by Swierstra [16] are essentially a functional programming application of the initial algebra semantics of inductive types. This consists of associating an inductive datatype to an endofunctor in a base category, then interpreting it as the initial object in the category of algebras determined by the functor [9, 19].

In its simplest form, taking sets (S) as the base category, each inductive datatype $\rho : \mathsf{S}$ can be associated with a covariant endofunctor (*signature functor*), i.e. a map $F : \mathsf{S} \to \mathsf{S}$ for which there exists a map (*functor map*) $\mathsf{fmap}_F\ \{A\ B\} :\ (A \to B) \to (F\ A \to F\ B)$ that preserves identities and composition, with $A, B : \mathsf{S}$ (always treated as implicit parameters). Semantically, an algebra determined by F (F*-algebra*) is a pair $\langle C, \phi \rangle$ where $C : \mathsf{S}$ is the *carrier* and $\phi : F\ C \to C$ is the *structure map*. $F\ C$ can be understood as the denotation of a grammar based on signature F, given carrier C. The initial object $\langle \mu F, \mathsf{in}_F \rangle$, where in_F is an isomorphism and thus has an inverse out_F, gives the denotation of ρ obtained as the fixpoint closure of F. In this way, the non-recursive structural characterisation of ρ, which essentially corresponds to case analysis, is separated from its recursive closure. For instance, in a functional language which allows

for datatype definitions with data constructors and Haskell-style destructors (while we mainly rely on Coq-style and standard algebraic notation), the following

$$\mathsf{dt\_def}\ \rho\ =\ \mathsf{c}_1\ (\tau_1[\rho/A])\mid\ldots\mid\mathsf{c}_k\ (\tau_k[\rho/A]) \tag{1}$$

can be decomposed in

$$\mathsf{dt\_def}\ F\ A\ =\ \mathsf{c}_1\ (\tau_1)\mid\ldots\mid\mathsf{c}_k\ (\tau_k) \tag{2}$$

and

$$\rho\ =_{df}\ \mathsf{Fix}\ F \tag{3}$$

where $\mathsf{Fix}\ F$ is the syntactic representation of μF, i.e.

$$\mathsf{dt\_def}\ \mathsf{Fix}\ F\ =\ \mathsf{in}\ (\mathsf{out}:\ F\ (\mathsf{Fix}\ F)) \tag{4}$$

For each F-algebra $\langle C,f\rangle$, the unique incoming algebra morphism from the initial algebra is determined by the unique *mediating map* $\mathsf{fold}_{F,C,f}:\mu F\to C$. Syntactically, this corresponds to the definition of $\mathsf{fold}\ F\ C:\ (F\ C\to C)\to(\mathsf{Fix}\ F\to C)$ as a recursive function.

$$\mathsf{fold}\ F\ C\ f\ x\ =_{df}\ f\ (\mathsf{fmap}\ F\ (\mathsf{fold}\ F\ C\ f)\ (\mathsf{out}\ x)) \tag{5}$$

Functors are composable by coproduct (+), i.e., if $F_1,F_2:\mathsf{S}\to\mathsf{S}$ are functors, so is $F_1{+}F_2$, with

$$\mathsf{dt\_def}\ (F_1{+}F_2)\ C\ =\ \mathsf{inl}\ (F_1\ C)\mid\mathsf{inr}\ (F_2\ C) \tag{6}$$

This results in a modular definition of the inductive datatype $\mathsf{Fix}\ (F_1{+}F_2)$ – not to be confused with $\mathsf{Fix}\ F_1{+}\mathsf{Fix}\ F_2$. In connection with coproducts, Haskell implementations of MDTs rely on type classes to automate injections and projections, using smart constructors and class constraints to express subsumption between functors. As a concrete example, following Swierstra [16], the conventional datatype

$$\mathsf{dt\_def}\ \mathsf{Trm}\ =\ \mathsf{lit}\ (\mathsf{Int})\mid\mathsf{add}\ (\mathsf{Trm}*\mathsf{Trm}) \tag{7}$$

can be decomposed into two modules

$$\mathsf{dt\_def}\ \mathsf{Trm}_{\mathsf{G1}}\ C\ =\ \mathsf{lit}\ (\mathsf{Int})\qquad\qquad\mathsf{dt\_def}\ \mathsf{Trm}_{\mathsf{G2}}\ C\ =\ \mathsf{add}\ (C*C) \tag{8}$$

and thus modularly defined:

$$\mathsf{Trm}_{\mathsf{G}}\ =_{df}\ \mathsf{Trm}_{\mathsf{G1}}+\mathsf{Trm}_{\mathsf{G2}}\qquad\qquad\mathsf{Trm}\ =_{df}\ \mathsf{Fix}\ \mathsf{Trm}_{\mathsf{G}} \tag{9}$$

Moreover, given a notion of value and a conventional recursive definition of evaluation

$$\begin{array}{ll}\mathsf{dt\_def}\ \mathsf{Val}\ =\ \mathsf{val}\ (\mathsf{vv}:\mathsf{Int}) & \mathsf{eval}:\mathsf{Trm}\to\mathsf{Val}\\ & \mathsf{eval}\ (\mathsf{lit}\ x)\ =_{df}\ \mathsf{val}\ x\\ & \mathsf{eval}\ (\mathsf{add}\ (e_1,e_2))\ =_{df}\ \mathsf{val}\ ((\mathsf{vv}\circ\mathsf{eval}\ e_1)+(\mathsf{vv}\circ\mathsf{eval}\ e_2))\end{array} \tag{10}$$

the latter can be represented by an algebra and modularly decomposed as follows, allowing for a modular definition of the dynamic semantics.

$$\begin{array}{ll}\mathsf{eval}_{\mathsf{G1}}:\mathsf{Trm}_{\mathsf{G1}}\ \mathsf{Val}\to\mathsf{Val} & \mathsf{eval}_{\mathsf{G1}}\ (\mathsf{lit}\ x)\ =_{df}\ \mathsf{val}\ x\\ \mathsf{eval}_{\mathsf{G2}}:\mathsf{Trm}_{\mathsf{G2}}\ \mathsf{Val}\to\mathsf{Val} & \mathsf{eval}_{\mathsf{G2}}\ (\mathsf{add}\ (x_1,x_2))\ =_{df}\ \mathsf{val}\ ((\mathsf{vv}\ x_1)+(\mathsf{vv}\ x_2))\\ \mathsf{eval}_{\mathsf{G}}:\mathsf{Trm}_{\mathsf{G}}\ \mathsf{Val}\to\mathsf{Val} & \mathsf{eval}_{\mathsf{G}}\ (\mathsf{inl}\ e)\ =_{df}\ \mathsf{eval}_{\mathsf{G1}}\ e\\ & \mathsf{eval}_{\mathsf{G}}\ (\mathsf{inr}\ e)\ =_{df}\ \mathsf{eval}_{\mathsf{G2}}\ e\end{array} \tag{11}$$

$$\mathsf{eval}\ e\ =_{df}\ \mathsf{fold}\ \mathsf{Trm}_{\mathsf{G}}\ \mathsf{Val}\ \mathsf{eval}_{\mathsf{G}}\ e \tag{12}$$

3 Impredicative encoding

The MDT representation discussed so far works well with Haskell, but not with Coq. Representing F as an inductive datatype is not problematic, but this is not so for the fixpoint closure. Since the constructor of $\mathsf{Fix}\,F$ has type $F\ (\mathsf{Fix}\ \mathsf{F}) \to \mathsf{Fix}\,F$, the datatype has a non-strictly positive occurrence in its definition, as parameter of the argument type – hence it is rejected by Coq. There is an analogous issue with the definition of fold, which is not structurally recursive. The solution to this problem adopted by Delaware *et al.* in [7], which we summarise here, goes back to Pfenning and Paulin-Mohring [13] in relying on a Church-style encoding of fixpoint operators, thus requiring impredicative definitions.

From the point of view of a type theoretic representation, the type of an algebra (that we may call *Church algebra*, or conventional algebra) can be identified with the type of its structure map.

$$\mathsf{Alg}^{\mathsf{C}}\ F\ C\ =_{df}\ F\ C \to C \tag{13}$$

If the initiality property of fixed points is weakened to an existence property, a fixpoint operator can be regarded as a function that maps an algebra to its carrier. An abstract definition of the type-level fixpoint operator $\mathsf{Fix}^{\mathsf{C}} : (\mathsf{S} \to \mathsf{S}) \to \mathsf{S}$ can then be given, as elimination rule for F-algebras, impredicatively with respect to S (this requires the impredicative set option in Coq, as used in MTC/3MT [7]).

$$\mathsf{Fix}^{\mathsf{C}}\ F =_{df} \forall A : \mathsf{S}.\ \mathsf{Alg}^{\mathsf{C}}\ F\ A \to A \tag{14}$$

The map $\mathsf{fold}^{\mathsf{C}}\ F\ C : \mathsf{Alg}^{\mathsf{C}}\ F\ C \to \mathsf{Fix}^{\mathsf{C}}\ F \to C$, corresponding to the elimination of a fixpoint value, can now be defined as the application of that value.

$$\mathsf{fold}^{\mathsf{C}}\ F\ C\ f\ x =_{df} x\ C\ f \tag{15}$$

Relying on the functoriality of F, the in-map $\mathsf{in}^{\mathsf{C}}\ F : F(\mathsf{Fix}\,F) \to \mathsf{Fix}\,F$ and the out-map $\mathsf{out}^{\mathsf{C}}\ F : \mathsf{Fix}\,F \to F(\mathsf{Fix}\,F)$ can be defined as functions.

$$\mathsf{in}^{\mathsf{C}}\ F\ =_{df}\ \lambda x\ A\ f.\ f(\mathsf{fmap}\ F\ (\mathsf{fold}^{\mathsf{C}}\ F\ A\ f)\ x) \tag{16}$$

$$\mathsf{out}^{\mathsf{C}}\ F\ =_{df}\ \mathsf{fold}^{\mathsf{C}}\ F\ (F(\mathsf{Fix}\,F))\ (\mathsf{fmap}\ F\ (\mathsf{in}^{\mathsf{C}}\ F)) \tag{17}$$

Notice that the definition of $\mathsf{fold}^{\mathsf{C}}\ F\ C\ f$ does not guarantee the uniqueness of the mediating map – it rather corresponds to a condition called quasi-initiality by Wadler [19]. In order to obtain uniqueness, hence to ensure that in^{C} is an isomorphism, the following implication needs to be proved for F [7, 11, 10].

$$(\forall x : \mathsf{Fix}^{\mathsf{C}}\ F.\ h\ (\mathsf{in}^{\mathsf{C}}\ F\ x)\ =\ f\ (\mathsf{fmap}\ F\ h\ x))\ \to\ (h\ =\ \mathsf{fold}^{\mathsf{C}}\ F\ C\ f) \tag{18}$$

Semantically, the impredicative encoding of the fixed points is closely associated with a constructor, usually called build, that allows for an alternative interpretation of inductive datatypes in terms of limit constructions, provably equivalent to the initial algebra semantics [8].

3.1 Indexed algebras

A relation can be represented as a function from the type of its tupled arguments to the type P of propositions. From the point of view of initial semantics, assuming P can be represented as a category, the modular representation of inductively defined relations only requires a shift of base category. Given a type K (i.e., $K : \mathsf{Type}$) and assuming it can be represented as a small category, we can take the category

of diagrams of type K in P as the base category for the relations of type $K \to \mathsf{P}$. In such category, an endofunctor $R : (K \to \mathsf{P}) \to (K \to \mathsf{P})$ that here we call *indexed functor*, is then associated with a map (*indexed functor map*) that preserves identities and composition.

$$\mathsf{fmap}^{\mathsf{I}}\ K\ R : \forall\ \{A\ B :\ K \to \mathsf{P}\}.\ (\forall w : K.\ A\ w \to B\ w) \to (\forall w : K.\ R\ A\ w \to R\ B\ w) \tag{19}$$

From the point of view of the impredicative encoding, an R-algebra can be characterised as an indexed map, given a carrier $D : K \to \mathsf{P}$.

$$\mathsf{Alg}^{\mathsf{CI}}\ K\ R\ D\ =_{df}\ \forall w : K.\ R\ D\ w \to D\ w \tag{20}$$

The corresponding fixpoint operator has type $((K \to \mathsf{P}) \to K \to \mathsf{P}) \to K \to \mathsf{P}$.

$$\mathsf{Fix}^{\mathsf{CI}}\ K\ R\ (w : K)\ =_{df}\ \forall A : K \to \mathsf{P}.\ \mathsf{Alg}^{\mathsf{CI}}\ K\ R\ A\ \to\ A\ w \tag{21}$$

The structuring operators can be defined as follows:

$$\mathsf{fold}^{\mathsf{CI}}\ K\ R :\ \forall A\ (f : \mathsf{Alg}^{\mathsf{CI}}\ K\ R\ A)\ (w : K).\ \mathsf{Fix}^{\mathsf{CI}}\ K\ R\ w \to A\ w\ =_{df}\ \lambda A\ f\ w\ e.\ e\ A\ f \tag{22}$$

$$\begin{array}{l}\mathsf{in}^{\mathsf{CI}}\ K\ R\ (w : K) : R\ (\mathsf{Fix}^{\mathsf{CI}}\ K\ R)\ w \to \mathsf{Fix}^{\mathsf{CI}}\ K\ R\ w\ =_{df} \\ \quad \lambda x\ A\ f.\ f\ w\ (\mathsf{fmap}^{\mathsf{I}}\ K\ R\ (\mathsf{fold}^{\mathsf{CI}}\ K\ R\ A\ f)\ w\ x)\end{array} \tag{23}$$

$$\begin{array}{l}\mathsf{out}^{\mathsf{CI}}\ K\ R\ (w : K) : \mathsf{Fix}^{\mathsf{CI}}\ K\ R\ w \to R\ (\mathsf{Fix}^{\mathsf{CI}}\ K\ R)\ w\ =_{df} \\ \quad \mathsf{fold}^{\mathsf{CI}}\ K\ R\ (R\ (\mathsf{Fix}^{\mathsf{CI}}\ K\ R))\ (\mathsf{fmap}^{\mathsf{I}}\ K\ R\ (\mathsf{in}^{\mathsf{CI}}\ K\ R))\ w\end{array} \tag{24}$$

3.2 Proof algebras

The impredicative encoding makes it comparatively easy to represent MDTs in Coq, but leaves us with the problem of how to reason inductively about them. Unlike the in-map of the categorical semantics, in^{C} is not a constructor – therefore, structural induction cannot be applied to a term of type $\mathsf{Fix}^{\mathsf{C}}\ F$. Let $P : T \to \mathsf{P}$ be a property and T the representation of an inductive datatype in the following goal, which we assume to be semantically provable by induction on T.

$$\Gamma, w : T \vdash g : P\ w \tag{25}$$

However, given $T =_{df} \mathsf{Fix}^{\mathsf{C}}\ F$ and the impredicative definition of $\mathsf{Fix}^{\mathsf{C}}$, the type T is not syntactically inductive, and no conventional induction principle can be applied. Nevertheless, we can prove

$$\forall v : T.\ \exists w : F\ T.\ P\ v\ =\ P\ (\mathsf{in}^{\mathsf{C}}\ F\ w) \tag{26}$$

as this follows from the equality $v\ =\ \mathsf{in}^{\mathsf{C}}\ F\ (\mathsf{out}^{\mathsf{C}}\ F\ v)$ which can be proved, provided $\mathsf{in}^{\mathsf{C}}\ F$ is shown to be an isomorphism – e.g., by proving (18). Rewriting (25) with (26), we obtain

$$\Gamma, w : F\ T \vdash g' : P\ (\mathsf{in}^{\mathsf{C}}\ F\ w) \tag{27}$$

Here it is possible to apply induction on w, since $F\ T$ is an inductive datatype: however, what we actually get is case analysis – the recursive arguments in $F\ T$ are hidden in the same sense as before, as they have type T rather than $F\ T$.

The solution adopted by Delaware *et al.* in [7], implemented in Coq and supported by MTC/3MT consists of packing an existential copy of the inductive term together with a proof that it satisfies the property, using Σ types. This involves replacing the conventional proof with one based on the representation of the goal as an algebra, i.e., a *proof algebra*.

$$\Gamma \vdash f : \ \mathsf{Alg}^{\mathsf{C}}\ F\ (\Sigma v.\ P\ v) \tag{28}$$

By folding such an algebra, one obtains

$$\Gamma, w : T \vdash \mathsf{fold}^{\mathsf{C}}\ F\ (\Sigma v.\ P\ v)\ f\ w : \ \Sigma v.\ P\ v \tag{29}$$

which states something weaker than the original goal (25). Nonetheless, under conditions associated with *well-formed proof algebras* in [7], (28) can be strengthened to (25). This technique is quite general, and it can be applied to inductive proofs in which the goals may depend on the inductive argument (i.e., it can deal with *dependent induction*). However, the proofs that are obtained in this way are essentially factored into two non-trivial parts – the application of a weak induction principle and a well-formedness proof – and therefore are quite different from conventional inductive ones.

3.3 Looking for a simpler solution

A natural question arises: is it possible to sacrifice some of the generality of the MTC approach, to obtain proofs that look more familiar? The whole point of using Σ types is to hide dependencies: a solution that does not involve them and so a positive answer to our question appear more feasible, when we can dispense with the use of dependent induction, by finding an alternative, equivalent formulation of the goal. In our schematic example (25) we get such reformulation, when we can find $S,\ Q : T \to \mathsf{P}$ and an indexed functor $R : (T \to \mathsf{P}) \to T \to \mathsf{P}$ such that $S =_{df} \mathsf{Fix}^{\mathsf{CI}}\ T\ R$, the following equivalence holds

$$\text{there exists } t \text{ s.t. } \ \Gamma \vdash t : \ \forall w : T.\ S\ w \to Q\ w \qquad \text{iff} \qquad \text{there exists } t' \text{ s.t. } \ \Gamma \vdash t' : \ \forall w : T.\ P\ w \tag{30}$$

and the following is semantically provable, as the new goal, by induction on h:

$$\Gamma,\ w : T,\ h : S\ w \vdash l : Q\ w \tag{31}$$

Intuitively, this means that the dependency of the proof on w can be lifted to a type dependency, given a sufficiently close analogy between T as modular inductive datatype and S as modular inductive predicate, therefore by rather using h of type $S\ w$ as inductive argument. Again, we need to expose the inductive structure by shifting to

$$\Gamma,\ w : T,\ h : R\ (\mathsf{Fix}^{\mathsf{CI}}\ T\ R)\ w \vdash l' : Q\ w \tag{32}$$

and this is not problematic. However, as before, we end up stuck with case analysis rather than proper induction. In order to solve this problem, we need to look at an alternative encoding of fixed points, based on Mendler-style induction [12, 1]. In fact, Mendler's approach makes it possible to build induction principles into impredicatively encoded fixed points. Notice that Mendler algebras are used by Delaware *et al.* [7], but have a different purpose there (i.e., controlling the order of evaluation), from the one we are proposing here.

4 Mendler algebras

We first present the Mendler-style semantics of inductive datatypes by introducing Mendler algebras as a category, following Uustalu and Vene [18]. Given a covariant functor $F : \mathsf{S} \to \mathsf{S}$, a Mendler algebra is a pair $\langle C, \Psi \rangle$ where $C : \mathsf{S}$ is the carrier and $\Psi\ A :\ (A \to C) \to (F\ A \to C)$, for each $A : \mathsf{S}$, is a map from morphisms to morphisms satisfying $\Psi\ A\ f\ =\ (\Psi\ C\ \mathsf{id}_C) \cdot (\mathsf{fmap}\ F\ f)$, with f a morphism from A to C. A morphism between Mendler algebras $\langle C_1, \Psi_1 \rangle$ and $\langle C_2, \Psi_2 \rangle$, is a morphism $h : C_1 \to C_2$ that satisfies $h \cdot \Psi_1\ C_1\ \mathsf{id}_{C_1} = \Psi_2\ C_1\ h$. The Mendler algebra semantics has been proved equivalent to the conventional one by Uustalu *et al.*. Assume F such that the conventional initial F-algebra $\langle \mu F, \mathsf{in}_F \rangle$ exists. Given the abbreviation

$$\mathsf{pre\_in}_F\ C\ (m : C \to \mu F)\ =_{df}\ \mathsf{in}_F \cdot (\mathsf{fmap}\ F\ m) :\ (F\ C \to \mu F) \tag{33}$$

we can prove the equation

$$\mathsf{in}_F\ =\ \mathsf{pre\_in}_F\ \mu F\ \mathsf{id} \tag{34}$$

by the isomorphic character of in_F. The Mendler algebra $\langle \mu F,\ \mathsf{pre\_in}_F \rangle$ can thus be shown to be the initial object in its category, and therefore used as alternative interpretation of the inductive datatype associated with F. For each Mendler algebra $\langle C, \Psi \rangle$, the unique incoming morphism from the initial Mendler F-algebra can be defined

$$\mathsf{mfold}\ F\ C\ \Psi\ x\ =_{df}\ \Psi\ (\mu F)\ (\mathsf{mfold}\ F\ C\ \Psi)\ (\mathsf{out}_F\ x) \tag{35}$$

Unlike the conventional fixpoint operator, the Mendler one can be encoded in Coq as an inductive datatype (though using the impredicative option).

$$\mathsf{dt\_def}\ \mathsf{MFix}\ F\ =\ \mathsf{pre\_in}\ (C : \mathsf{S})\ (b : C \to \mathsf{MFix}\ F)\ (c : F\ C) \tag{36}$$

However in, as defined by equation (34) in this setting, is still not a constructor, and the definition of mfold is not structurally recursive. Therefore, also in this case, it seems more convenient to resort to an impredicative encoding, following [12, 7].

4.1 Impredicative Mendler algebra encoding

Mendler algebras can be characterised impredicatively by the type of their structure maps, and a fixpoint operator can be defined as in the conventional case [12, 7].

$$\mathsf{Alg}^{\mathsf{M}}\ F\ C\ =_{df}\ \forall A.\ (A \to C) \to (F\ A \to C) \tag{37}$$

$$\mathsf{Fix}^{\mathsf{M}}\ F\ =_{df}\ \forall C.\ \mathsf{Alg}^{\mathsf{M}}\ F\ C \to C \tag{38}$$

Unlike the conventional case, the type of a Mendler algebra can be read as specification of an iteration step, where the bound type variable A represents the type of the recursive calls. The corresponding fold operator

$$\mathsf{fold}^{\mathsf{M}}\ F\ C\ f\ x\ =_{df}\ x\ C\ f \tag{39}$$

indeed has type

$$\mathsf{fold}^{\mathsf{M}}\ F\ C :\ (\forall A.\ (A \to C) \to (F\ A \to C)) \to (\mathsf{Fix}^{\mathsf{M}}\ \mathsf{F}) \to C \tag{40}$$

which can represent an induction principle, under the assumption that the argument to the induction hypothesis is only used therein without further analysis [12, 1]. In-maps and out-maps can be defined as follows

$$\mathsf{in}^{\mathsf{M}}\ F\ (x : F(\mathsf{Fix}^{\mathsf{M}}\ F)) : \mathsf{Fix}^{\mathsf{M}}\ F\ =_{df}\ \lambda A\ (f : \mathsf{Alg}^{\mathsf{M}}\ F\ A).\ f\ (\mathsf{Fix}^{\mathsf{M}}\ F)\ (\mathsf{fold}^{\mathsf{M}}\ F\ A\ f)\ x \tag{41}$$

$$\begin{array}{l}\mathsf{out}^{\mathsf{M}}\ F\ (x : \mathsf{Fix}^{\mathsf{M}}\ F) : F\ (\mathsf{Fix}^{\mathsf{M}}\ F)\ =_{df}\ x\ (F\ (\mathsf{Fix}^{\mathsf{M}}\ F)) \\ \qquad (\lambda A\ (r : A \to F\ (\mathsf{Fix}^{\mathsf{M}}\ F))\ (a : F\ A).\ \mathsf{fmap}\ F\ (\lambda y : A.\ \mathsf{in}^{\mathsf{M}}\ F\ (r\ y))\ a)\end{array} \tag{42}$$

As in the conventional case, impredicative fixpoint definitions give us quasi-initiality. The uniqueness condition of $\mathsf{fold}^{\mathsf{M}}\ F\ A\ f$ that is needed for initiality, in a way which parallels (18), is given by

$$(\forall x : F\ (\mathsf{Fix}^{\mathsf{M}}\ F).\ h\ (\mathsf{in}^{\mathsf{M}}\ F\ x)\ =\ f\ (\mathsf{Fix}^{\mathsf{M}}\ F)\ h\ x)\ \to\ h\ =\ \mathsf{fold}^{\mathsf{M}}\ F\ A\ f \tag{43}$$

to be proven for a fixed F, for every $A : \mathsf{S}$, $f : \mathsf{Alg}^{\mathsf{M}}\ F\ A$ and $h : \mathsf{Fix}^{\mathsf{M}}\ F \to A$ [18].

4.2 Indexed Mendler algebras

As before, we need indexed algebras to deal with relations. The definitions are similar to the conventional ones, with K a type, $R : (K \to \mathsf{P}) \to (K \to \mathsf{P})$ an indexed functor, and $D : K \to \mathsf{P}$ an indexed carrier.

$$\mathsf{Alg}^{\mathsf{MI}}\ K\ R\ D =_{df} \forall A.\ (\forall w : K.\ A\ w \to D\ w) \to \forall w : K.\ R\ A\ w \to D\ w \tag{44}$$

$$\mathsf{Fix}^{\mathsf{MI}}\ K\ R\ w =_{df} \forall A.\ \mathsf{Alg}^{\mathsf{MI}}\ K\ R\ A \to A\ w \tag{45}$$

$$\mathsf{fold}^{\mathsf{MI}}\ K\ R\ D\ (f : \mathsf{Alg}^{\mathsf{MI}}\ K\ R\ D)\ (w : K)\ (x : \mathsf{Fix}^{\mathsf{MI}}\ K\ R\ w) =_{df} x\ D\ f \tag{46}$$

$$\begin{array}{l}\mathsf{in}^{\mathsf{MI}}\ K\ R\ (w : K)\ (x : R\ (\mathsf{Fix}^{\mathsf{MI}}\ K\ R)\ w) : \mathsf{Fix}^{\mathsf{MI}}\ K\ R\ w\ =_{df} \\ \qquad \lambda A\ (f : \mathsf{Alg}^{\mathsf{MI}}\ K\ R\ A).\ f\ (\mathsf{Fix}^{\mathsf{MI}}\ K\ R)\ (\mathsf{fold}^{\mathsf{MI}}\ K\ R\ A\ f)\ w\ x\end{array} \tag{47}$$

$$\begin{array}{l}\mathsf{out}^{\mathsf{MI}}\ K\ R\ (w : K)\ (x : \mathsf{Fix}^{\mathsf{MI}}\ K\ R\ w) : R\ (\mathsf{Fix}^{\mathsf{MI}}\ K\ R)\ w\ = \\ \qquad x\ (R\ (\mathsf{Fix}^{\mathsf{MI}}\ K\ R))\ (\lambda\ A\ (r : \forall v.\ A\ v \to R\ (\mathsf{Fix}^{\mathsf{MI}}\ K\ R)\ v) \\ \qquad\quad (w : K)\ (a : R\ A\ w).\ \mathsf{fmap}^{\mathsf{I}}\ R\ (\lambda y : A\ w.\ \mathsf{in}^{\mathsf{MI}}\ K\ R\ w\ (r\ w\ y))\ a)\end{array} \tag{48}$$

As an example, we can define inductively a relation $\mathsf{Eval} : (\mathsf{Trm} * \mathsf{Val}) \to \mathsf{P}$ that agrees with eval.

$$\begin{array}{l}\mathsf{dt\_def}\ \mathsf{Eval}_{\mathsf{G}}\ (A : (\mathsf{Trm} * \mathsf{Val}) \to \mathsf{P})\ :\ (\mathsf{Trm} * \mathsf{Val}) \to \mathsf{P}\ = \\ \qquad \mathsf{ev1} : \forall x : \mathsf{Int}.\ \mathsf{Eval}_{\mathsf{G}}\ A\ (\mathsf{lit}\ x, \mathsf{val}\ x) \\ \qquad \mathsf{ev2} : \forall e_1\ e_2 : \mathsf{Trm}, x_1\ x_2 : \mathsf{Val}.\ A(e_1,x_1)\ \wedge\ A(e_2,x_2) \to \\ \qquad\qquad\qquad\qquad \mathsf{Eval}_{\mathsf{G}}\ A\ (\mathsf{add}(e_1,e_2), \mathsf{val}((\mathsf{vv}\ x_1) + (\mathsf{vv}\ x_2)))\end{array} \tag{49}$$

$$\mathsf{Eval}\ =_{df}\ \mathsf{Fix}^{\mathsf{MI}}\ (\mathsf{Trm} * \mathsf{Val})\ \mathsf{Eval}_{\mathsf{G}} \tag{50}$$

4.3 Proof algebras, Mendler-style

Reconsider the schematic example in Section 3.2: the problem in (32) was the missing induction hypothesis, that cannot be obtained by appealing to the standard inductive principle, as the recursive occurrences are wrapped in a non-inductive type. Intuitively, this can be fixed by giving such an hypothesis explicitly. This would give us a generic representation of the step lemma in our inductive proof.

$$\Gamma,\ h_0 : \forall v : T.\ \mathsf{Fix}^{\mathsf{CI}}\ T\ R\ v \to Q\ v,\ w : T,\ h_1 : R\ (\mathsf{Fix}^{\mathsf{CI}}\ T\ R)\ w\ \vdash\ q : Q\ w \tag{51}$$

However, here the type of h_0 is actually too specific to be that of the induction hypothesis with respect to h_1 – as a result, the sequent is too weak to take us to the main goal (31). At this point, Mendler's intuition comes into play: under the assumption that the argument passed to the induction hypothesis is used only there, without further case analysis, and that therefore we make no use of its type structure, its type can be represented by a fresh type variable – the key feature of Mendler-style induction [12, 1]. We can then strengthen (51) to the following, more abstract goal.

$$\Gamma,\ A : \mathsf{Type},\ h_0 : \forall v : T.\ A\ v \to Q\ v,\ w : T,\ h_1 : R\ A\ w\ \vdash\ p : Q\ w \tag{52}$$

Given $f =_{df} \lambda A\ h_0\ w\ h_1.\ p$, the above is equivalent to

$$\Gamma\ \vdash\ f :\ \mathsf{Alg}^{\mathsf{MI}}\ T\ R\ Q \tag{53}$$

Now we have an indexed Mendler algebra. The original goal, equivalent to (25) by a reformulation of (30) with $S = \mathsf{Fix}^{\mathsf{MI}}\ T\ R$, can then be obtained by folding, without need of further adjustments.

$$\Gamma\ \vdash\ \mathsf{fold}^{\mathsf{MI}}\ T\ R\ Q\ f\ :\ \forall w : T.\ S\ w \to Q\ w \tag{54}$$

In order to prove (52), case analysis (as provided in Coq e.g. by *inversion* and *destruct* tactics [3]) can be applied to h_1, allowing us to reason on the structure of $R\ A\ w$. This actually results in doing induction on that structure, as the induction hypothesis h_0 is already there. In this way, we can minimise the overhead of combining inductive proofs with modular datatypes. Proving an inductive lemma boils down to constructing the appropriate Mendler algebra – the rest is either conventional, or comes for free. In connection with MDT, such algebras can be regarded as proof modules, that can be composed together in the usual sense of case analysis on coproducts [16, 7], in the same straightforward way as evaluation algebras (the original motivating example by Swierstra [16]). This sounds attractive, from the point of view of the applications in which the relational aspect is predominant, such as structural operational semantics.

4.4 Problematic aspects

Which could be the downsides of the Mendler-based approach? As already observed, relying on impredicative encodings gives us for free only a weak semantics of inductive datatypes, i.e., a quasi-initial one. However, initiality is needed virtually everywhere in our proofs, to ensure in-maps and out-maps are inverses, i.e.

$$(A)\ \ \mathsf{out}^{\mathsf{M}}\ F\ (\mathsf{in}^{\mathsf{M}}\ F\ x)\ =\ x \qquad\qquad (B)\ \ \mathsf{in}^{\mathsf{M}}\ F\ (\mathsf{out}^{\mathsf{M}}\ F\ x)\ =\ x \tag{55}$$

and similarly for the indexed case. In order to get proper initial semantics, functor-specific proofs of properties such as (43) for base category S, or the corresponding one for $K \to \mathsf{P}$, need to be carried out.

This may be regarded as a general weakness of impredicative approaches including MTC/3MT [7, 6], as remarked by Keuchel and Schrijvers [11]. Nonetheless, in discussing the well-formedness of Church encodings [7], Delaware *et al.* argue that dealing with this issue is not too hard, as indeed MTC provides automation for doing so.

A more specific problem is related to the iterative character of Mendler-style recursion, and correspondingly, to the non-dependent character of Mendler-style induction. Mendler algebras make it possible to factor induction into case analysis and folding, but this restricts induction, in the sense of what is called *Mendler iteration* by Abel, Matthes and Uustalu [1]: the argument of the induction hypothesis cannot be used anywhere else, effectively ruling out dependent induction. This means there are problems that cannot be solved in their original form. As an example, MTC [7] proves the type soundness of a language with a dynamic semantics that is recursively defined as a total evaluation function. This problem can be reformulated with respect to our concrete example in Section 2, using our definition of eval (12).

$$\Gamma, e : \mathsf{Trm}, t : \mathsf{Typ} \vdash k : \mathsf{TypOf}\ (e,\ t) \to \mathsf{TypOf}\ (\mathsf{lit} \circ \mathsf{vv}\ (\mathsf{eval}\ e),\ t) \tag{56}$$

Using the MTC approach, (56) can be proved by dependent induction on the structure of term e. Given $\mathsf{dt\_def}\ \mathsf{Typ} = \mathsf{N}$ and assuming for simplicity TypOf is a conventional inductive predicate

$$\begin{array}{ll} \mathsf{dt\_def} & \mathsf{TypOf} : \mathsf{Trm} * \mathsf{Typ} \to \mathsf{P} = \\ & \mathsf{tof1} : \forall v : \mathsf{Val}.\ \mathsf{TypOf}\ (\mathsf{lit} \circ \mathsf{vv}\ v,\ \mathsf{N}) \\ & \mathsf{tof2} : \forall e_1\ e_2 : \mathsf{Trm}.\ \mathsf{TypOf}\ (e_1, \mathsf{N}) \wedge \mathsf{TypOf}\ (e_2, \mathsf{N}) \to \mathsf{TypOf}\ (\mathsf{add}(e_1, e_2), \mathsf{N}) \end{array} \tag{57}$$

the proof is ultimately based on a proof algebra of type $\mathsf{Alg}^{\mathsf{C}}\ \mathsf{Trm}_{\mathsf{G}}\ (\Sigma e.\ \forall t : \mathsf{Typ}.\ \mathsf{TypOf}\ (e,\ t) \to \mathsf{TypOf}\ (\mathsf{lit} \circ \mathsf{vv}\ (\mathsf{eval}\ e),\ t))$, although as already noticed, folding this algebra only gives us the backbone of the whole proof.

This is not possible using our Mendler-style approach, as we cannot deal with the dependency of the goal on the inductive argument e. What we can do instead, is to rely on the relational formulation of evaluation given by Eval (50), which can be shown to satisfy (30), and prove

$$\Gamma, e : \mathsf{Trm}, v : \mathsf{Val},\ t : \mathsf{Typ},\ h : \mathsf{Eval}\ (e, v) \vdash l : \mathsf{TypOf}\ (e,\ t) \to \mathsf{TypOf}\ (\mathsf{lit} \circ \mathsf{vv}\ v,\ t) \tag{58}$$

reasoning by induction on the structure of Eval. This reformulation of the goal essentially matches (31). In this case, a proof can be obtained by simply folding an indexed Mendler algebra of type $\mathsf{Alg}^{\mathsf{MI}}\ (\mathsf{Trm} * \mathsf{Val})\ \mathsf{Eval}_{\mathsf{G}}\ (\lambda(e, v).\ \forall t : \mathsf{Typ}.\ \mathsf{TypOf}\ (e,\ t) \to \mathsf{TypOf}\ (\mathsf{lit} \circ \mathsf{vv}\ v,\ t))$, which provides our instance of (53).

An alternative way to obtain a relational equivalent of (56) is to lift the modular datatype Trm to a modular predicate $\mathsf{IsTrm} : (\mathsf{Trm}_{\mathsf{G}}\ \mathsf{Trm}) \to \mathsf{P}$, with $\mathsf{IsTrm} =_{df} \mathsf{Fix}^{\mathsf{MI}}\ (\mathsf{Trm}_{\mathsf{G}}\ \mathsf{Trm})\ \mathsf{IsTrm}_{\mathsf{G}}$, where

$$\begin{array}{lll} \mathsf{dt\_def}\ \mathsf{IsTrm}_{\mathsf{G}}\ A & = & \mathsf{isLit} : \forall x : \mathsf{Int}.\ \mathsf{IsTrm}_{\mathsf{G}}\ A\ (\mathsf{lit}\ x) \\ & & \mid \mathsf{isAdd} : \forall e_1\ e_2 : \mathsf{Trm}.\ A\ e_1 \wedge A\ e_2 \to \mathsf{IsTrm}_{\mathsf{G}}\ A\ (\mathsf{add}\ (e_1, e_2)) \end{array} \tag{59}$$

and then prove

$$\Gamma, e : \mathsf{Trm}, w : \mathsf{IsTrm}\ e, t : \mathsf{Typ} \vdash k : \mathsf{TypOf}\ (e,\ t) \to \mathsf{TypOf}\ (\mathsf{lit} \circ \mathsf{vv}\ (\mathsf{eval}\ e),\ t) \tag{60}$$

reasoning by Mendler induction on w. Notice that eval in the MTC example [7] is actually defined as the fold of a Mendler algebra, rather than a conventional one, in order to allow for control over the evaluation order – this is related to the form of their semantics though, and completely unrelated to our use of Mendler-style induction.

5 Case study

The use of relational formulations appears particularly natural in specifications based on small-step rules in the style of SOS, originally introduced by Plotkin [14]. Yet in order to formulate each relation modularly, we need to build encodings based on functors that reflect the structure of those relations. This inevitably makes things more complex, especially when we have to deal with mutually inductive definitions. In order to test the applicability of Mendler proof algebras to the formalisation of a semantic framework, we have formalised a language $\mathscr{L}$ with a comparatively rich syntactic structure, including types (Typ), patterns (Pat), declarations (Dec) and expressions (Exp), as well as value environments ($\mathsf{Env}^{\mathsf{E}}$) and typing environments ($\mathsf{Env}^{\mathsf{T}}$). We rely on SOS to give a partial specification of the language: partial, insofar as we do not specify any behaviour in case of pattern matching failure – therefore, we cannot prove type soundness, which in fact does not hold. However, we can still prove type preservation – and this suffices for us, as an example of the structural complexity we are aiming at.

The full language specification is available with the Coq formalisation in the companion code at [17]. Here we outline the specification using conventional dataytpes. The Coq formalisation is entirely based on modular datatypes, although for simplicity we rely on monolithic functors (we have not yet implemented the smart constructor mechanism that facilitates the use of coproducts).

$$\begin{array}{llll} \mathsf{dt\_def} & \mathsf{Typ} & = & \mathsf{ty}(\mathsf{Id}^{\mathsf{T}}) \mid \mathsf{Typ} \Rightarrow \mathsf{Typ} \mid \mathsf{type\_env}(\mathsf{Env}^{\mathsf{T}}) \\ \mathsf{dt\_def} & \mathsf{Pat} & = & \mathsf{vr}^{\mathsf{p}}(\mathsf{Id},\mathsf{Typ}) \mid \mathsf{cn}^{\mathsf{p}}(\mathsf{Id},\mathsf{Typ}) \mid \mathsf{apply}^{\mathsf{p}}(\mathsf{Pat},\mathsf{Pat}) \\ \mathsf{dt\_def} & \mathsf{Dec} & = & \mathsf{env}(\mathsf{Env}^{\mathsf{E}}) \mid \mathsf{match}(\mathsf{Pat},\mathsf{Exp}) \mid \mathsf{join}(\mathsf{Dec},\mathsf{Dec}) \\ \mathsf{dt\_def} & \mathsf{Exp} & = & \mathsf{vr}(\mathsf{Id}) \mid \mathsf{cn}(\mathsf{Id},\mathsf{Typ}) \mid \mathsf{closure}(\mathsf{Env}^{\mathsf{E}},\mathsf{Pat},\mathsf{Exp}) \\ & & & \mid \mathsf{apply}(\mathsf{Exp},\mathsf{Exp}) \mid \mathsf{scope}(\mathsf{Dec},\mathsf{Exp}) \end{array} \tag{61}$$

$$\mathsf{Env}\,A =_{df} \mathsf{Id} \to \mathsf{option}\,A \qquad \mathsf{Env}^{\mathsf{T}} =_{df} \mathsf{Env}\;\mathsf{Typ} \qquad \mathsf{Env}^{\mathsf{E}} =_{df} \mathsf{Env}\;\mathsf{Exp} \tag{62}$$

The language $\mathscr{L}$ is based on simply typed lambda calculus with pattern matching and first class environments. We use two sets of identifiers – Id^{T} for type variables and Id for object variables and constants. Constants and pattern variables are annotated with types. $\Rightarrow$ is the usual function type constructor. We use closures instead of lambda abstractions to ensure values are closed terms and avoid dealing with substitution. Abstraction is defined over patterns (rather than simply over variables). Matching patterns with expressions give declarations, which may evaluate to environments. Declarations can be joined together and used in scope expressions. Values can be specified as follows.

$$\begin{array}{lll} \text{Data values :} & h \in & \mathsf{cn}(x,\tau) \mid \mathsf{apply}(h,v) \\ \text{Values :} & v \in & \mathsf{closure}(\rho,p,e) \mid h \end{array} \tag{63}$$

The typing relations have the following signatures. Notice that patterns and values can be typed in a context-free way, unlike expressions and declarations.

$$\begin{array}{ll} \text{Patterns :} & \mathsf{TypOPat} \;:\; \mathsf{Pat} * \mathsf{Typ} \to \mathsf{P} \\ \text{Environments :} & \mathsf{TypOEnv} \;:\; \mathsf{Env}^{\mathsf{E}} * \mathsf{Env}^{\mathsf{T}} \to \mathsf{P} \\ \text{Declarations :} & \mathsf{TypODec} \;:\; \mathsf{Env}^{\mathsf{T}} * \mathsf{Dec} * \mathsf{Typ} \to \mathsf{P} \\ \text{Expressions :} & \mathsf{TypOExp} \;:\; \mathsf{Env}^{\mathsf{T}} * \mathsf{Exp} * \mathsf{Typ} \to \mathsf{P} \end{array} \tag{64}$$

The transition relations have the following signatures.

$$\begin{array}{ll} \text{Declarations :} & \mathsf{DecStep} \;:\; \mathsf{Env}^{\mathsf{E}} * \mathsf{Dec} * \mathsf{Dec} \to \mathsf{P} \\ \text{Expressions :} & \mathsf{ExpStep} \;:\; \mathsf{Env}^{\mathsf{E}} * \mathsf{Exp} * \mathsf{Exp} \to \mathsf{P} \end{array} \tag{65}$$

Expressions and declarations may depend on each other, and therefore can only have a mutually inductive definition. Analogously, the definitions of the typing relations and of the transition relations for these two syntactic categories involve mutual induction. Therefore we need to introduce functors to reason about mutually inductively defined sets, as well as mutually inductively defined relations.

5.1 Mutually inductive sets

Two mutually recursive datatypes in the base category S, can be represented in terms of bi-functors $F_1,\ F_2 :\ \mathsf{S}*\mathsf{S}\to\mathsf{S}$, where bi-functoriality is expressed as existence of a map $\mathsf{fmap}^{\mathsf{D}}$ which satisfies the appropriate form of the usual preservation properties.

$$\mathsf{fmap}^{\mathsf{D}} :\ \forall\ \{A_1\ A_2\ B_1\ B_2 : \mathsf{S}\}\ (f_1 : A_1 \to B_1)\ (f_2 : A_2 \to B_2).\ F\ (A_1,A_2) \to F\ (B_1,B_2) \tag{66}$$

$$\begin{array}{l} \mathsf{fmap}^{\mathsf{D}}\ g_1\ g_2\ (\mathsf{fmap}^{\mathsf{D}}\ f_1\ f_2)\ =\ \mathsf{fmap}^{\mathsf{D}}\ (g_1 \cdot f_1)\ (g_2 \cdot f_2) \\ \mathsf{fmap}^{\mathsf{D}}\ \mathsf{id}_A\ \mathsf{id}_B\ =\ \mathsf{id}_{FAB} \end{array} \tag{67}$$

The definitions of Mendler bi-algebra, fixpoint and fold operators can be given using pairs.

$$\begin{array}{l} \mathsf{Alg}^{\mathsf{D}}\ (F_1,F_2)\ (C_1,C_2)\ =_{df}\ (\forall A_1\ A_2.\ (A_1 \to C_1) \to (A_2 \to C_2) \to F_1\ (A_1,A_2) \to C_1, \\ \qquad\qquad \forall A_1\ A_2.\ (A_1 \to C_1) \to (A_2 \to C_2) \to F_2\ (A_1,A_2) \to C_2) \end{array} \tag{68}$$

$$\begin{array}{l} \mathsf{Fix}^{\mathsf{D}}\ (F_1,F_2)\ =_{df}\ (\forall A_1\ A_2.\ \mathsf{Alg}^{\mathsf{D}}\ (F_1,F_2)\ (A_1,A_2) \to A_1, \\ \qquad\qquad \forall A_1\ A_2.\ \mathsf{Alg}^{\mathsf{D}}\ (F_1,F_2)\ (A_1,A_2) \to A_2) \end{array} \tag{69}$$

$$\begin{array}{l} \mathsf{fold}_1^{\mathsf{D}}\ (F_1,F_2)\ (C_1,C_2)\ (f : \mathsf{Alg}^{\mathsf{D}}\ (F_1,F_2)\ (C_1,C_2)) : \\ \qquad\qquad \mathsf{fst}\ (\mathsf{Fix}^{\mathsf{D}}\ (F_1,F_2)) \to C_1\ =_{df}\ \lambda e.\ e\ C_1\ C_2\ f \end{array} \tag{70}$$

$$\begin{array}{l} \mathsf{fold}_2^{\mathsf{D}}\ (F_1,F_2)\ (C_1,C_2)\ (f : \mathsf{Alg}^{\mathsf{D}}\ (F_1,F_2)\ (C_1,C_2)) : \\ \qquad\qquad \mathsf{snd}\ (\mathsf{Fix}^{\mathsf{D}}\ (F_1,F_2)) \to C_2\ =_{df}\ \lambda e.\ e\ C_1\ C_2\ f \end{array} \tag{71}$$

All the syntactic categories of $\mathscr{L}$ can then be represented as MDTs, using bi-functors for mutually defined Decl and Exp.

$$\begin{array}{l} \mathsf{dt\_def}\ \mathsf{Typ}_{\mathsf{G}}\ T\ =\ \mathsf{ty}(\mathsf{Id}^{\mathsf{T}}) \mid T \Rightarrow T \mid \mathsf{type\_env}\ (\mathsf{Env}^{\mathsf{T}}\ T) \qquad\qquad \mathsf{Typ}\ =_{df}\ \mathsf{Fix}^{\mathsf{M}}\ \mathsf{Typ}_{\mathsf{G}} \\ \mathsf{dt\_def}\ \mathsf{Pat}_{\mathsf{G}}\ P\ =\ \mathsf{vr}^{\mathsf{P}}(\mathsf{Id},T) \mid \mathsf{cn}^{\mathsf{P}}(\mathsf{Id},T) \mid \mathsf{apply}^{\mathsf{P}}(P,P) \qquad\qquad \mathsf{Pat}\ =_{df}\ \mathsf{Fix}^{\mathsf{M}}\ \mathsf{Pat}_{\mathsf{G}} \\ \mathsf{dt\_def}\ \mathsf{Dec}_{\mathsf{G}}\ D\ E\ =\ \mathsf{env}(\mathsf{Env}\ E) \mid \mathsf{match}(\mathsf{Pat},E) \mid \mathsf{join}(D,D) \\ \mathsf{dt\_def}\ \mathsf{Exp}_{\mathsf{G}}\ D\ E\ =\ \mathsf{vr}(\mathsf{Id}) \mid \mathsf{cn}(\mathsf{Id},\mathsf{Typ}) \mid \mathsf{closure}(\mathsf{Env}\ E,\mathsf{Pat},E)\ \mid \mathsf{apply}(E,E) \mid \mathsf{scope}(D,E) \\ \qquad \mathsf{Dec}\ =_{df}\ \mathsf{fst}\ (\mathsf{Fix}^{\mathsf{D}}\ (\mathsf{Dec}_{\mathsf{G}},\mathsf{Exp}_{\mathsf{G}})) \qquad\qquad \mathsf{Exp}\ =_{df}\ \mathsf{snd}\ (\mathsf{Fix}^{\mathsf{D}}\ (\mathsf{Dec}_{\mathsf{G}},\mathsf{Exp}_{\mathsf{G}})) \end{array} \tag{72}$$

5.2 Mutually inductive relations

Given types K_1,K_2, two mutually recursive relations depending on such types in base categories $K_1 \to \mathsf{P}$, $K_2 \to \mathsf{P}$, can be represented by indexed bi-functors R_1,R_2, with

$$R_1\ K_1 : (K_1 \to \mathsf{P}) * (K_2 \to \mathsf{P}) \to (K_1 \to \mathsf{P}) \qquad R_2\ K_1 : (K_1 \to \mathsf{P}) * (K_2 \to \mathsf{P}) \to (K_2 \to \mathsf{P}) \tag{73}$$

characterised by maps

$$\begin{aligned}&\mathsf{fmap}_1^{\mathsf{H}}\ (K_1,K_2)\ R_1 :\ \forall\ \{A_1\ A_2 :\ K_1 \to \mathsf{P}\}\ \{B_1\ B_2 :\ K_2 \to \mathsf{P}\}.\\ &\quad (\forall w : K_1.\ A_1\ w \to B_1\ w) \to (\forall w : K_2.\ A_2\ w \to B_2\ w) \to\\ &\quad\quad \forall w : K_1.\ R_1\ (A_1,A_2)\ w \to R_1\ (B_1,B_2)\ w\end{aligned} \tag{74}$$

$$\begin{aligned}&\mathsf{fmap}_2^{\mathsf{H}}\ (K_1,K_2)\ R_2 :\ \forall\ \{A_1\ A_2 :\ K_1 \to \mathsf{P}\}\ \{B_1\ B_2 :\ K_2 \to \mathsf{P}\}.\\ &\quad (\forall w : K_1.\ A_1\ w \to B_1\ w) \to (\forall w : K_2.\ A_2\ w \to B_2\ w) \to\\ &\quad\quad \forall w : K_2.\ R_2\ (A_1,A_2)\ w \to R_2\ (B_1,B_2)\ w\end{aligned} \tag{75}$$

Given carriers $D_1 : K_1 \to \mathsf{P}$, $D_2 : K_2 \to \mathsf{P}$, we can now define indexed Mendler bi-algebras and the associated notions (see [17] for more details).

$$\begin{aligned}&\mathsf{Alg}^{\mathsf{H}}\ (K_1,K_2)\ (R_1,R_2)\ (D_1,D_2)\ =_{df}\\ &\quad (\forall A_1\ A_2.\ (\forall w : K_1.\ A_1\ w \to D_1\ w) \to (\forall w : K_2.\ A_2\ w \to D_2\ w) \to\\ &\quad\quad \forall w : K_1.\ R_1\ (A_1,A_2)\ w \to D_1\ w,\\ &\quad \forall A_1\ A_2.\ (\forall w : K_1.\ A_1\ w \to D_1\ w) \to (\forall w : K_2.\ A_2\ w \to D_2\ w) \to\\ &\quad\quad \forall w : K_2.\ R_2\ (A_1,A_2)\ w \to D_2\ w)\end{aligned} \tag{76}$$

$$\begin{aligned}&\mathsf{Fix}^{\mathsf{H}}\ (K_1,K_2)\ (R_1,R_2)\ =_{df}\\ &\quad (\lambda w : K_1.\ \forall A_1\ A_2.\ \mathsf{Alg}^{\mathsf{H}}\ (K_1,K_2)\ (R_1,R_2)\ (A_1,A_2) \to A_1\ w,\\ &\quad \lambda w : K_2.\ \forall A_1\ A_2.\ \mathsf{Alg}^{\mathsf{H}}\ (K_1,K_2)\ (R_1,R_2)\ (A_1,A_2) \to A_2\ w)\end{aligned} \tag{77}$$

$$\begin{aligned}&\mathsf{fold}_1^{\mathsf{H}}\ (K_1,K_2)\ (R_1,R_2)\ (D_1,D_2)\ (f : \mathsf{Alg}^{\mathsf{H}}\ (K_1,K_2)\ (R_1,R_2)\ (D_1,D_2))\ (w : K_1) :\\ &\quad \mathsf{fst}\ (\mathsf{Fix}^{\mathsf{H}}\ (K_1,K_2)\ (R_1,R_2))\ w \to D_1\ w\ =_{df}\ \lambda w\ e.\ e\ D_1\ D_2\ f\end{aligned} \tag{78}$$

$$\begin{aligned}&\mathsf{fold}_2^{\mathsf{H}}\ (K_1,K_2)\ (R_1,R_2)\ (D_1,D_2)\ (f : \mathsf{Alg}^{\mathsf{H}}\ (K_1,K_2)\ (R_1,R_2)\ (D_1,D_2))\ (w : K_2) :\\ &\quad \mathsf{snd}\ (\mathsf{Fix}^{\mathsf{H}}\ (K_1,K_2)\ (R_1,R_2))\ w \to D_2\ w\ =_{df}\ \lambda w\ e.\ e\ D_1\ D_2\ f\end{aligned} \tag{79}$$

While the typing relations for patterns TypOPat can be represented modularly using an indexed functor and $\mathsf{Fix}^{\mathsf{I}}$, the corresponding relations for declarations and expressions, i.e. TypODec and TypOExp respectively, are mutually defined and therefore need to be represented as indexed bi-functors closed by $\mathsf{Fix}^{\mathsf{H}}$. Such is also the case for DecStep and ExpStep, which can be defined as follows, given the corresponding indexed bi-functors $\mathsf{DecStep}_{\mathsf{G}} : (\mathsf{Env}^{\mathsf{E}} * \mathsf{Dec} * \mathsf{Dec} \to \mathsf{P},\ \mathsf{Env}^{\mathsf{E}} * \mathsf{Exp} * \mathsf{Exp} \to \mathsf{P}) \to \mathsf{Env}^{\mathsf{E}} * \mathsf{Dec} * \mathsf{Dec} \to \mathsf{P}$, and $\mathsf{ExpStep}_{\mathsf{G}} : (\mathsf{Env}^{\mathsf{E}} * \mathsf{Dec} * \mathsf{Dec} \to \mathsf{P},\ \mathsf{Env}^{\mathsf{E}} * \mathsf{Exp} * \mathsf{Exp} \to \mathsf{P}) \to \mathsf{Env}^{\mathsf{E}} * \mathsf{Exp} * \mathsf{Exp} \to \mathsf{P}$.

$$\mathsf{DecStep}\ =_{df}\ \mathsf{fst}\ (\mathsf{Fix}^{\mathsf{H}}\ (\mathsf{Env}^{\mathsf{E}} * \mathsf{Dec} * \mathsf{Dec},\ \mathsf{Env}^{\mathsf{E}} * \mathsf{Exp} * \mathsf{Exp})\ (\mathsf{DecStep}_{\mathsf{G}},\ \mathsf{ExpStep}_{\mathsf{G}})) \tag{80}$$

$$\mathsf{ExpStep}\ =_{df}\ \mathsf{snd}\ (\mathsf{Fix}^{\mathsf{H}}\ (\mathsf{Env}^{\mathsf{E}} * \mathsf{Dec} * \mathsf{Dec},\ \mathsf{Env}^{\mathsf{E}} * \mathsf{Exp} * \mathsf{Exp})\ (\mathsf{DecStep}_{\mathsf{G}},\ \mathsf{ExpStep}_{\mathsf{G}})) \tag{81}$$

5.3 Type preservation

Type preservation in $\mathscr{L}$ can be expressed as follows

$$\begin{aligned}&\Gamma, \rho : \mathsf{Env}^{\mathsf{E}} \vdash\ (\forall (d_1\ d_2 : \mathsf{Dec}).\ \mathsf{DecStep}\ (\rho, d_1, d_2) \to \mathsf{DecTSafe}\ (\rho, d_1, d_2)\\ &\quad\quad \wedge\ (\forall (e_1\ e_2 : \mathsf{Exp}).\ \mathsf{ExpStep}\ (\rho, e_1, e_2) \to \mathsf{ExpTSafe}\ (\rho, e_1, e_2)\end{aligned} \tag{82}$$

where

$$\begin{array}{l}\mathsf{DecTSafe}\ (\rho,d_1,d_2) =_{df}\ \forall(t:\mathsf{Typ})\ (\gamma:\mathsf{Env}^{\mathsf{T}}).\\ \qquad \mathsf{TypOEnv}\ (\rho,\gamma) \to \mathsf{TypODec}\ (\gamma,d_1,t) \to \mathsf{TypODec}\ (\gamma,d_2,t)\\ \mathsf{ExpTSafe}\ (\rho,e_1,e_2) =_{df}\ \forall(t:\mathsf{Typ})\ (\gamma:\mathsf{Env}^{\mathsf{T}}).\\ \qquad \mathsf{TypOEnv}\ (\rho,\gamma) \to \mathsf{TypOExp}\ (\gamma,e_1,t) \to \mathsf{TypOExp}\ (\gamma,e_2,t)\end{array} \tag{83}$$

The context Γ includes premises of shape

$$(IN\,x\ =\ IN\,y)\ \to\ (x\ =\ y) \tag{84}$$

where IN is the in-map for one of the datatypes – such premises can be discharged when the corresponding initiality conditions (43) are proven. It also includes premises of shape

$$\forall x : D_G, IsD_G\,x. \tag{85}$$

where D_G is the unfolding of a modular datatype D, and IsD_G is the unfolding of a modular predicate IsD that represents the relational lifting of D, in the sense of our example (59). Such premises are needed, as the proof involves sublemmas that are proved by induction on the syntactic categories – and so, for instance, $\mathsf{Typ_G}\ \mathsf{Typ}$ has to be lifted to $\mathsf{IsTyp_G} : (\mathsf{Typ_G}\ \mathsf{Typ} \to \mathsf{P}) \to \mathsf{Typ_G}\ \mathsf{Typ} \to \mathsf{P}$.

Crucially, the pair of DecTSafe and ExpTSafe can be a carrier for the indexed bi-functor determined by DecStep and ExpStep. In order to prove type preservation by mutual induction on the structure of DecStep and ExpStep, we define an indexed Mendler bi-algebra that has $(\mathsf{DecTSafe}, \mathsf{ExpTSafe})$ as indexed carrier, where the index types are $\mathsf{Env^E} * \mathsf{Dec} * \mathsf{Dec}$ and $\mathsf{Env^E} * \mathsf{Exp} * \mathsf{Exp}$

$$\begin{array}{l}\mathsf{TPAlg}\ =_{df}\ \mathsf{Alg^H}\ (\mathsf{Env^E} * \mathsf{Dec} * \mathsf{Dec},\ \mathsf{Env^E} * \mathsf{Exp} * \mathsf{Exp})\\ \qquad\qquad (\mathsf{DecStep_G},\ \mathsf{ExpStep_G})\ (\mathsf{DecTSafe},\ \mathsf{ExpTSafe})\end{array} \tag{86}$$

After finding proofs $f_1 : \mathsf{fst}\ \mathsf{TPAlg}$ and $f_2 : \mathsf{snd}\ \mathsf{TPAlg}$, we can construct a proof of (82) by applying to them $\mathsf{fold}_1^{\mathsf{H}}$ and $\mathsf{fold}_2^{\mathsf{H}}$, respectively (see [17] for details).

6 Conclusion

Motivated by the importance of modularity in program development, semantics and verification, we have discussed the use of MDTs, their semantic foundations and their impredicative encoding along the lines of existing work [7, 11, 16]. We have shown how impredicative MDT encodings based on Mendler algebras can be used to reason about inductively defined relations, in a way that is comparatively close to a more conventional style of reasoning based on closed datatypes, by providing a simpler notion of proof algebra, if less general, than the one proposed by Delaware *et al.* [7]. Our approach can be regarded as a novel application of Mendler-style induction [12, 1, 18], as well as a technique that could be integrated in existing frameworks based on the impredicative encoding, such as MTC/3MT [7, 6]. Mendler's original insight [12] was in the semantics of inductive datatypes – the case made here, is for using that insight as a modular proof technique. From the point of view of possible applications to semantics and verification in frameworks such as OTT [15], the relational style that can be supported seems to fit in well with SOS and in particular with component-based approaches, such as the one proposed by Churchill, Mosses, Sculthorpe and Torrini [4]. Our plans for future work include integrating our technique in MTC/3MT, and comparing this approach with the container-based one proposed by Keuchel and Schrijvers [11].

Acknowledgments: We thank Steven Keuchel, Neil Sculthorpe, Casper Bach Poulsen and the anonymous reviewers for feedback on earlier versions, and members of the Theory Group at Swansea University, including Peter Mosses, Anton Setzer and Ulrich Berger, for discussion. The writing of this paper has been supported by EU funding (H2020 FET) to KU Leuven for the GRACEFUL project. Preliminary work was funded by the EPSRC grant (EP/I032495/1) to Swansea University for the PLANCOMPS project.

References

[1] A. Abel, R. Matthes & T. Uustalu (2005): *Iteration and coiteration schemes for higher-order and nested datatypes. Theor. Comput. Sci.* 333(1-2), pp. 3–66, doi:10.1016/j.tcs.2004.10.017.

[2] H. Barendregt (1992): *Lambda Calculi with Types*. In: *Hand. of Logic in Co. Sc.*, Oxford, pp. 117–309.

[3] Y. Bertot & P. Casteran (2004): *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, doi:10.1007/978-3-662-07964-5.

[4] M. Churchill, P. D. Mosses, N. Sculthorpe & P. Torrini (2015): *Reusable Components of Semantic Specifications*. In: *TAOSD 12*, LNCS 8989, Springer, pp. 132–179, doi:10.1007/978-3-662-46734-3_4.

[5] T. Coquand & G. Huet (1988): *The calculus of constructions*. *Information and Computation* 76, pp. 95–120, doi:10.1016/0890-5401(88)90005-3.

[6] B. Delaware, S. Keuchel, T. Schrijvers & B. C.d.S. Oliveira (2013): *Modular Monadic Meta-theory*. In: *ICFP'13*, ACM, pp. 319–330, doi:10.1145/2500365.2500587.

[7] B. Delaware, B. C. d. S. Oliveira & T. Schrijvers (2013): *Meta-theory à la carte*. In: *Proc. POPL '13*, pp. 207–218, doi:10.1145/2429069.2429094.

[8] N. Ghani, T. Uustalu & V. Vene (2004): *Build, Augment and Destroy, Universally*. In: *Proc. APLAS '04*, pp. 327–347, doi:10.1007/978-3-540-30477-7_22.

[9] T. Hagino (1987): *A Typed Lambda Calculus with Categorical Type Constructors*. In: *Category Theory and Computer Science*, pp. 140–157, doi:10.1007/3-540-18508-9_24.

[10] G. Hutton (1999): *A Tutorial on the Universality and Expressiveness of Fold*. *J. Funct. Program.* 9(4), pp. 355–372, doi:10.1017/S0956796899003500.

[11] S. Keuchel & T. Schrijvers (2013): *Generic Datatypes à la Carte*. In: *9th ACM SIGPLAN Workshop on Generic Programming (WGP)*, pp. 1–11, doi:10.1145/2502488.2502491.

[12] N. P. Mendler (1991): *Inductive Types and Type Constraints in the Second-Order lambda Calculus*. *Ann. Pure Appl. Logic* 51(1-2), pp. 159–172, doi:10.1016/0168-0072(91)90069-X.

[13] F. Pfenning & C. Paulin-Mohring (1989): *Inductively Defined Types in the Calculus of Constructions*. In: *Math. Foundations of Programming Semantics*, pp. 209–228, doi:10.1007/BFb0040259.

[14] G. D. Plotkin (2004): *A structural approach to operational semantics*. *J. Log. Algebr. Program.* 60-61, pp. 17–139, doi:10.1016/j.jlap.2004.03.009.

[15] P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar & R. Strniša (2010): *Ott: Effective Tool Support for the Working Semanticist*. *Journal of Functional Programming* 20(1), pp. 71–122, doi:10.1017/S0956796809990293.

[16] W. Swierstra (2008): *Data types à la carte*. *Journal of Functional Programming* 18(4), pp. 423–436, doi:10.1017/S0956796808006758.

[17] P. Torrini (2015): *Language specification and type preservation proofs in Coq – companion code*. Available at `http://cs.swan.ac.uk/~cspt/MDTC`.

[18] T. Uustalu & V. Vene (1999): *Mendler-Style Inductive Types, Categorically*. *Nord. J. Comput.* 6(3), p. 343.

[19] P. Wadler (1990): *Recursive types for free!* Available at `http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`.