New Equations for Neutral Terms

A Sound and Complete Decision Procedure, Formalized

Guillaume Allais Conor McBride
University of Strathclyde
{guillaume.allais, conor.mcbride}@strath.ac.uk

Pierre Boutillier
PPS - Paris Diderot
pierre.boutillier@pps.univ-paris-diderot.fr

Abstract

The definitional equality of an intensional type theory is its test of type compatibility. Today's systems rely on ordinary evaluation semantics to compare expressions in types, frustrating users with type errors arising when evaluation fails to identify two 'obviously' equal terms. If only the machine could decide a richer theory! We propose a way to decide theories which supplement evaluation with 'ν-rules', rearranging the neutral parts of normal forms, and report a successful initial experiment.

We study a simple λ-calculus with primitive fold, map and append operations on lists and develop in Agda a sound and complete decision procedure for an equational theory enriched with monoid, functor and fusion laws.

Keywords Normalization by Evaluation, Logical Relations, Simply-Typed Lambda Calculus, Map Fusion

1. Introduction

The programmer working in intensional type theory is no stranger to 'obviously true' equations she wishes held *definitionally* for her program to typecheck without having to chase down ill-typed terms and brutally coerce them. In this article, we present one way to relax definitional equality, thus accommodating some of her longings. We distinguish three types of fundamental relations between terms.

The first denotes computational rules: it is untyped, *oriented* and denoted by $\rightsquigarrow$ in its one step version or $\rightsquigarrow^\star$ when the reflexive transitive congruence closure is considered. In Table 1, we introduce a few such rules which correspond to the equations the programmer writes to define functions. They are referred to as δ (for *definitions*) and ι (for pattern-matching on *inductive* data) rules and hold computationally just like the more common β-rule.

The second is the judgmental equality ($\equiv$): it is typed, tractable for a machine to decide and typically includes η-rules for negative types therefore internalizing some kind of *extensionality*. Table 2 presents such rules, explaining that some types have unique constructors which the programmer can demand. They are well supported in e.g. Epigram [15] and Agda [35] both for functions and records but still lacking for records in Coq [28].

The third is the propositional equality ($=$): this lets us state and give evidence for equations on open terms which may not be identified judgmentally. Table 3 shows a kit for building computationally inert *neutral* terms growing layers of thwarted progress around a variable which we dub the 'nut', together with some equations on neutral terms which held only propositionally – until now. This paper shows how to extend the judgmental equality with these new 'ν-rules'. We gain, for example, that `map swap . map swap` $\equiv$ `id`, where `swap` swaps the elements of a pair.

```
map : (a → b) → list a → list b
map f []        ⇝ []
map f (x :: xs) ⇝ f x :: map f xs

(++) : list a → list a → list a
[]      ++ ys ⇝ ys
x :: xs ++ ys ⇝ x :: (xs ++ ys)

fold : (a → b → b) → b → list a → b
fold c n []        ⇝ n
fold c n (x :: xs) ⇝ c x (fold c n xs)
```

Table 1. $\delta\iota$-rules - computational

```
Γ ⊢ f ≡ λ x. f x       : a → b
Γ ⊢ p ≡ (π₁ p , π₂ p)  : a * b
Γ ⊢ u ≡ ()             : 1
```

Table 2. η-rules - canonicity

$$\boxed{\texttt{x}}\quad \boxed{\square\ \texttt{a}}\quad \boxed{\pi_1\ \square}\quad \boxed{\pi_2\ \square}\quad \boxed{\square\ \texttt{++ ys}}\quad \boxed{\texttt{map f}\ \square}\quad \boxed{\texttt{fold n c}\ \square}$$

$$\begin{array}{rcl}
\boxed{\boxed{\texttt{xs}}\ \texttt{++ []}} & = & \boxed{\texttt{xs}} \\
\boxed{\boxed{(\boxed{\texttt{xs}}\ \texttt{++ ys})}\ \texttt{++ zs}} & = & \boxed{\boxed{\texttt{xs}}\ \texttt{++ (ys ++ zs)}} \\
\boxed{\texttt{map id}\ \boxed{\texttt{xs}}} & = & \boxed{\texttt{xs}} \\
\boxed{\texttt{map f}\ \boxed{(\texttt{map g}\ \boxed{\texttt{xs}})}} & = & \boxed{\texttt{map (f . g)}\ \boxed{\texttt{xs}}} \\
\boxed{\texttt{map f}\ \boxed{(\boxed{\texttt{xs}}\ \texttt{++ ys})}} & = & \boxed{\boxed{\texttt{map f}\ \boxed{\texttt{xs}}}\ \texttt{++ map f ys}} \\
\boxed{\texttt{fold c n}\ \boxed{(\texttt{map f}\ \boxed{\texttt{xs}})}} & = & \boxed{\texttt{fold (c . f) n}\ \boxed{\texttt{xs}}} \\
\boxed{\texttt{fold c n}\ \boxed{(\boxed{\texttt{xs}}\ \texttt{++ ys})}} & = & \boxed{\texttt{fold c (fold c n ys)}\ \boxed{\texttt{xs}}}
\end{array}$$

Table 3. ν-rules

A ν-rule is an equation between neutral terms with the same nut which holds just by structural induction on the nut, with $\beta\delta\iota$ reducing subgoals to inductive hypotheses – the classic proof pattern of Boyer and Moore [14]. Consequently, we need only use ν-rules to standardize neutral terms after ordinary evaluation stops. This separability makes implementation easy, but the proof of its completeness correspondingly difficult. Here, we report a successful experiment in formalizing a modified normalization by evaluation proof for simply-typed λ-calculus with list primitives and the ν-rules above.

Contents We define the terms of the theory and deliver a sound and complete normalization algorithm in Sections 2 to 5. We then explain how this promising experiment can be scaled up to type theory (Section 6) thus suggesting that other frustrating equations of a similar character may soon come within our grasp (Section 7).

2. Our Experimental Setting

In a dependently-typed setting, one has to deal with issues unrelated to the matter at hand: Danielsson's formalization of a Type Theory as an inductive-recursive family uses a non strictly positive datatype [21], Abel et al. [2] resort to recursive domain equations together with logical relations proving them meaningful, McBride's proposition [32] is only able to steal the judgmental equality of the implementation language and Chapman's big step formulation is not proven terminating [17].

We propose a preliminary experiment on a calculus for which the formalization in Agda is tractable: we are interested in the modifications to be made to an existing implementation in order to get a complete procedure for the extended equational theory. We developed the algorithm during Boutillier's internship at Strathclyde [13]; Allais completed the formalized meta-theory.

Types The set of types is parametrized by a finite set of base types $`\alpha_1, \ldots, `\alpha_n$ it can build upon. These unanalysed base types give us a simple way to model expressions exhibiting some parametric polymorphism.

$$\sigma, \tau, \ldots ::= \ `\alpha_k \mid `\mathtt{1} \mid \sigma \mathbin{`\times} \tau \mid \sigma \mathbin{`\to} \tau \mid `\mathtt{list}\ \sigma$$

Remark In the Agda implementation this indexing by a finite set of base types is modelled by defining a nat-indexed family $\mathtt{type}_n$ with a constructor $`\alpha$ taking a natural number k bounded by n (an element of $\mathtt{Fin}\ n$) to refer to the k^{th} base type.

Terms Terms follow the grammar presented below and the typing rules described in Figure 1 where contexts are just snoc lists of variable names together with their type.

$$\begin{aligned} t, u, \ldots ::= \ & x \mid `\lambda\, x.t \mid t \mathbin{`\$} u \mid `\langle\rangle \mid t \mathbin{`,} u \mid `\pi_1\, t \mid `\pi_2\, t \mid `[] \\ & \mid hd \mathbin{`::} tl \mid `\mathtt{map}(f, xs) \mid xs \mathbin{`{+\!\!+}} ys \mid `\mathtt{fold}(c, n, xs) \end{aligned}$$

For sake of clarity in the formalization, we quote the constructors of our object language, making a clear distinction from the corresponding features of the host language, Agda, where we use the standard 'typed de Bruijn index' representation of well-typed terms [8, 23] to eliminate junk from consideration. In our treatment here, we always assume freshness of the variables introduced by λ-abstractions. And we do not artificially separate well-typed terms and typing derivations; in other words we will use alternatively $\Gamma \vdash t : \sigma$ and $t \colon \Gamma \vdash \sigma$ to denote the same objects.

Weakening The notion of context inclusion gives rise to a weakening operation $\mathtt{wk\_}$ which can be viewed as the action on morphisms of the functor $\_ \vdash \sigma$ from the category of contexts and their inclusions to the category of well-typed terms and functions between them. It is defined inductively (cf. Figure 1) rather than as a function transporting membership predicates from one context to its extension in order to avoid having to use an extensionality axiom to prove two context inclusion proofs to be the same. This more intensional presentation can already be found under the name *order preserving embeddings* in Chapman's thesis [17].

From types to contexts We can lift the notion of well-typed terms $\Gamma \vdash \sigma$ to whole parallel substitutions. For any two contexts named Γ and Δ, the well-typed parallel substitution from Γ to Δ is defined by:

$$\Delta \vDash^s \Gamma = \begin{cases} \top & \text{if } \Gamma = \varepsilon \\ \Delta \vDash^s \Gamma' \times \Delta \vdash \sigma & \text{if } \Gamma = \Gamma' \cdot (x : \sigma) \end{cases}$$

We write $t[\rho]$ for the application of the parallel substitution $\rho \colon \Delta \vDash^s \Gamma$ to the term $t \colon \Gamma \vdash \sigma$ yielding a term of type $\Delta \vdash \sigma$.

Remark All the notions described in this document can be lifted in a pointwise fashion to either contexts when they are defined on types or parallel substitutions when they deal with terms. We will assume these extensions defined and casually use the same name (augmented with: ^{s}) for the extension and the original concept.

Judgmental Equality The equational theory of the calculus, denoted $\equiv_{\beta\delta\iota\eta\nu}$, is quite naturally the congruence closure of the $\beta\delta\iota\eta\nu$-rules described earlier where reductions under λ-abstraction are allowed. In this paper, we also mention the relation $\rightsquigarrow^{*}_{\beta\delta\iota\eta\nu}$ where the rules presented earlier are all considered with a left to right orientation (except for the identity laws for the list functor and the list monoid) thus inducing a notion of *reduction*. The soundness theorem proves that not only is the term produced by our normalization procedure related to the source one but it is a reduct of it.

One easy sanity check we recommend before starting to work on the meta-theory was to give a shallow embedding of the calculus in a pre-existing sound type theory and to show that the reduction relation is compatible with the propositional equality in this theory. We used Agda extended with a postulate stating extensional equality for non-dependent functions in our formalization. Once the reader is convinced that no silly mistakes were made in the equational theory, she can start the implementation.

3. Reduction Machinery

When looking in details at different accounts of normalization by evaluation [4, 12, 18, 19], the reader should be able to detect that there are two phases in the process: firstly the evaluation function building elements of the model from well-typed terms performs $\beta\delta\iota$-reductions and does not reduce under λ-abstractions effectively building closures – using the λ-abstractions of the host language – when encountering one. Secondly the quoting machinery extracting terms from the model performs η-expansions where needed which will cause the closures to be reduced and new computations to be started. This two-step process was already more or less present in Berger and Schwichtenberg's original paper [12]:

> Obviously each term in β-normal form may be transformed into long β-normal form by suitable η-expansions. Therefore each term r may be transformed into a unique long β-normal form $r^\star$ by β-conversion and η-expansions.

Building on this ascertainment, we construct a three (rather than two) staged process successively performing $\beta\delta\iota$, η and finally ν reductions whilst always potentially calling back a procedure from a preceding stage to reduce further non-normal terms appearing when e.g. going under λ-abstractions during η-expansion, distributing a map over an append, etc.

3.1 The Three Stages of Standardization

The normalization and standardization process goes through three successive stages whence the need to define three different subsets

$$\frac{}{\texttt{base}:\ \varepsilon \subseteq \varepsilon} \qquad \frac{pr : \Gamma \subseteq \Delta}{\texttt{pop!}\ pr : \Gamma \cdot (x : \sigma) \subseteq \Delta \cdot (x : \sigma)} \qquad \frac{pr : \Gamma \subseteq \Delta}{\texttt{step}\ pr : \Gamma \subseteq \Delta \cdot (x : \sigma)}$$

$$\frac{(x : \sigma) \in \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\Gamma \cdot (x : \sigma) \vdash t : \tau}{\Gamma \vdash `\lambda\, x.t : \sigma\ `\!\!\to \tau} \qquad \frac{\Gamma \vdash t : \sigma\ `\!\!\to \tau \quad \Gamma \vdash u : \sigma}{\Gamma \vdash t\ `\$\ u : \tau}$$

$$\frac{}{\Gamma \vdash `\langle\rangle : `1} \qquad \frac{\Gamma \vdash t : \sigma \quad \Gamma \vdash u : \tau}{\Gamma \vdash t\ `,\ u : \sigma\ `\!\!\times \tau} \qquad \frac{\Gamma \vdash t : \sigma\ `\!\!\times \tau}{\Gamma \vdash `\pi_1\, t : \sigma} \qquad \frac{\Gamma \vdash t : \sigma\ `\!\!\times \tau}{\Gamma \vdash `\pi_2\, t : \tau}$$

$$\frac{}{\Gamma \vdash `[]\ : `\texttt{list}\ \sigma} \qquad \frac{\Gamma \vdash hd : \sigma \quad \Gamma \vdash tl : `\texttt{list}\ \sigma}{\Gamma \vdash hd\ `\!\!::\ tl : `\texttt{list}\ \sigma}$$

$$\frac{\Gamma \vdash xs : `\texttt{list}\ \sigma \quad \Gamma \vdash ys : `\texttt{list}\ \sigma}{\Gamma \vdash xs\ `\!\!+\!\!+\ ys : `\texttt{list}\ \sigma} \qquad \frac{\Gamma \vdash f : \sigma\ `\!\!\to \tau \quad \Gamma \vdash xs : `\texttt{list}\ \sigma}{\Gamma \vdash `\texttt{map}(f, xs) : `\texttt{list}\ \tau}$$

$$\frac{\Gamma \vdash c : \sigma\ `\!\!\to \tau\ `\!\!\to \tau \quad \Gamma \vdash n : \tau \quad \Gamma \vdash xs : `\texttt{list}\ \sigma}{\Gamma \vdash `\texttt{fold}(c, n, xs) : \tau}$$

Figure 1: Context inclusion and typing rules

of terms of our calculus. They have to be understood simply as syntactic category restricting the shape of terms typed in the same way as the ones in the original languages except for the few extra constructors for which we explicitly detail what they mean.

Remark It should be noted that the two last steps never reduce a term to a constructor-headed one for datatypes (lists in our setting). In particular, the last step only rearranges stuck terms to produce terms which are themselves stuck. In other words: if a term (a list in our case) is convertible to a constructor headed term (be it either nil or cons), then it is reduced to it by the first step of the reduction.

Example We will consider the normalization of $(`\lambda\, x.x)\ `\$(`\lambda\, x.x)$ of type $\varepsilon \vdash `\texttt{list}\ (`1\ `\!\!\times `\alpha_k)\ `\!\!\to `\texttt{list}\ (`1\ `\!\!\times `\alpha_k)$ as a running example demonstrating the successive steps.

Untyped $\beta\iota$-reductions The first intermediate language we are going to encounter is composed of weak-head $\beta\delta\iota$-normal expressions i.e. we never reduce under a lambda, this role being assigned to the η-expansion routine. Having λ-closures as first-class values is one of the characteristics of this approach.

$$\begin{aligned}
m &::= x \mid m\ `\$\ w \mid `\pi_1\, m \mid `\pi_2\, m \mid `\texttt{fold}(w_1, w_2, m) \\
&\quad\ \mid `\texttt{map}(w, m) \mid m\ `\!\!+\!\!+\ w \\
w &::= m \mid `\lambda[\rho]x.t \mid `\langle\rangle \mid w_1\ `,\ w_2 \mid `[] \mid w_1\ `\!\!::\ w_2 \\
\rho &::= \varepsilon \mid \rho, x \mapsto w
\end{aligned}$$

Figure 2: Weak-head normal forms

These values are computed using a simple off the shelf environment machine which returns a constructor when facing one; stores the evaluation environment in a λ-closure when evaluating a term starting with a $`\lambda$; and calls an helper function (e.g. $\texttt{wh-}\$\$$, $\texttt{wh-}\pi_1$, $\texttt{wh-}\pi_2$, etc.) on the recursively evaluated subterms when uncovering an eliminator. These helper functions either return a neutral if the interesting subterm was stuck or perform the elimination which may start new computations (e.g. in the application case). We call `wh-norm` this evaluation function.

Remark This reduction step is absolutely type-agnostic and could therefore be performed on terms devoid of any type information as in e.g. Coq where conversion is untyped. Keeping and propagating *some* types (e.g. the codomain of the function in a map) is nonetheless needed to be able to infer back the type of the whole expression which is crucial in the following steps.

Example The untyped evaluation reduces our simple example $(`\lambda\, x.x)\ `\$(`\lambda\, x.x)$ to the usual identity function: $`\lambda[\,\texttt{tt}\,]x.x$.

Type-directed η-expansion Then an η-expansion step kicks in and produces η-long values in a type-directed way. It insists that the only neutrals worthy of being considered normal forms are the ones of the base type. It also carves out the subset of stuck lists in a separate syntactic category l thus preparing for the last step which will leave most of the rest of the language untouched.

$$\begin{aligned}
n &::= x \mid n\ `\$\ v \mid `\pi_1\, n \mid `\pi_2\, n \mid `\texttt{fold}(v_1, v_2, l) \\
v &::= n_{`\alpha_k} \mid l \mid `\lambda\, x.v \mid `\langle\rangle \mid v_1\ `,\ v_2 \mid `[] \mid v_1\ `\!\!::\ v_2 \\
l &::= n_{`\texttt{list}\ \sigma} \mid `\texttt{map}(v, l) \mid l\ `\!\!+\!\!+\ v
\end{aligned}$$

Figure 3: η-long values

The η-expansion of product and function type actually calls back the subroutines for $\beta\delta\iota$-rules projecting components out of pairs or performing function application – here to the variable newly introduced. This step is the only one requiring a name generator which allows us to avoid threading such an artifact along the whole reduction machinery. We call $\eta\texttt{norm}$ the main function performing this step and present it in Figure 4. $\eta\texttt{list}$ and $\eta\texttt{neut}$ are two trivial auxiliary functions going structurally through either lists or neutral terms and calling $\eta\texttt{norm}$ whenever necessary.

$$\begin{aligned}
\eta\texttt{norm}(`\alpha_k)\ t &= \eta\texttt{neut}\ t \\
\eta\texttt{norm}(`\texttt{list}\ \sigma)\ t &= \eta\texttt{list}\ \sigma\ t \\
\eta\texttt{norm}(`1)\ t &= `\langle\rangle \\
\eta\texttt{norm}(\sigma\ `\!\!\times \tau)\ t &= \eta\texttt{norm}\ \sigma\ (\texttt{wh-}\pi_1\, t)\ `,\ \eta\texttt{norm}\ \tau\ (\texttt{wh-}\pi_2\, t) \\
\eta\texttt{norm}(\sigma\ `\!\!\to \tau)\ t &= `\lambda\, x.\, \eta\texttt{norm}\, \tau (t\ \texttt{wh-}\$\$\ x))
\end{aligned}$$

Figure 4: From weak-head normal forms to η-long ones

Example The η-expansion of the evaluated form $`\lambda[\,\texttt{tt}\,]x.x$ of type $\varepsilon \vdash `\texttt{list}\ (`1\ `\!\!\times `\alpha_k)\ `\!\!\to `\texttt{list}\ (`1\ `\!\!\times `\alpha_k)$ proceeds in multiple steps.

- The arrow type forces us to introduce a λ-abstraction: $`\lambda\, x.\, \eta\texttt{norm}\ (`\texttt{list}\ (`1\ `\!\!\times `\alpha_k))\ ((`\lambda[\,\texttt{tt}\,]x.x)\ \texttt{wh-}\$\$\ x)$.
- Now, $(`\lambda[\,\texttt{tt}\,]x.x)\ \texttt{wh-}\$\$\ x$ trivially reduces to $x$, a neutral of list type, left unmodified by $\eta$-expansion. Hence the $\eta$-long form: $`\lambda\, x.x$.

***ν*-rules reorganizing neutrals** Standard forms have a very specific shape due to the fact that we now completely internalize the ν-rules. The new constructor $\texttt{`map}\langle \_ \, , \, \_ \rangle\texttt{`++} \; \_$ – referred to as "mapp" – has the obvious semantics that it represents the concatenation of a stuck map and a list.

$$
\begin{aligned}
n &::= \; x \mid n \,\texttt{`\$}\, v \mid \texttt{`}\pi_1\, n \mid \texttt{`}\pi_2\, n \mid \texttt{`fold}(v_1, v_2, n) \\
v &::= \; n_{\texttt{`}\alpha_k} \mid s \mid \texttt{`}\lambda\, x.v \mid \texttt{`}\langle\rangle \mid v_1 \;\texttt{`,}\; v_2 \mid \texttt{`[]} \mid v_1 \;\texttt{`::}\; v_2 \\
s &::= \; \texttt{`map}\langle v_1 \, , \, n \rangle\texttt{`++}\; v_2
\end{aligned}
$$

Figure 5: Standard Forms

The standard lists s are produced by flattening the stuck map / append trees present in l after the end of the previous procedure whilst the fold / map and fold / append fusion rules are applied in order to compute folds further and reach the point where a stuck fold is stuck on a *real* neutral lists. These reductions are computed by the mutually defined `nf-norm`, `nf-neut` and `nf-list` respectively turning η-long normals, neutrals and lists into elements of the corresponding standard classes. `nf-norm` and `nf-neut` are mostly structural except for the few cases described in Figure 6.

We define `standard` as being the composition of η`norm` and `nf-norm` whilst norm is the composition of `wh-norm` and `standard`. As one can see below, ν-rules can restart computations in subterms by invoking subroutines of the evaluation function `wh-norm`. Formally proving the termination of the whole process is therefore highly non-trivial.

$$
\begin{aligned}
&\texttt{nf-norm}(\texttt{`list}\;\sigma) xs_{ne} = \texttt{nf-list}\; xs \\
&\texttt{nf-neut}(\texttt{`fold}\; c\; n\; xs) = \texttt{nf-fold}\; c\; n\; (\texttt{nf-list}\; xs)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{nf-list}\; xs_{ne} &= \texttt{`map}\langle \mathrm{norm}(\texttt{`}\lambda\, x.x) \, , \, xs \rangle\texttt{`++}\; \texttt{`[]} \\
\texttt{nf-list}\; (\texttt{`map}\; f\, xs) &= \texttt{nf-map}\; f\; (\texttt{nf-list}\; xs) \\
\texttt{nf-list}\; (xs \;\texttt{`++}\; ys) &= \texttt{nf-++}(\texttt{nf-list}\; xs)(\texttt{nf-norm}\; \_\; ys)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{nf-fold}\; c\; n\; (\texttt{`map}\langle f \, , \, xs \rangle\texttt{`++}\; ys) = \texttt{`fold}\; cf\; ih\; xs \\
&\quad \text{where} \quad cf = \texttt{standard}\; \_\; (c\; \texttt{wh-}\infty\; f) \\
&\quad \phantom{\text{where}} \quad ih = \texttt{standard}\; \_\; (\texttt{wh-fold}\; c\; n\; ys)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{nf-map}\; f\; (\texttt{`map}\langle g \, , \, xs \rangle\texttt{`++}\; ys) = \texttt{`map}\langle fg \, , \, xs \rangle\texttt{`++}\; fys \\
&\quad \text{where} \quad fg = \texttt{standard}\; \_\; (f\; \texttt{wh-}\infty\; g)) \\
&\quad \phantom{\text{where}} \quad fys = \texttt{standard}\; \_\; (\texttt{wh-map}\; f\; ys)
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{nf-++}(\texttt{`map}\langle f \, , \, xs \rangle\texttt{`++}\; ys)\; zs = \texttt{`map}\langle f \, , \, xs \rangle\texttt{`++}\; yzs \\
&\quad \text{where} \quad yzs = \texttt{standard}\; \_\; (ys\; \texttt{wh-++}\; zs)
\end{aligned}
$$

Figure 6: From η-long values to standard ones

Example `nf-norm` does not touch the λ-abstraction but expands the neutral x of type $\texttt{`list}\; (\texttt{`1} \;\texttt{`}\!\times\; \texttt{`}\alpha_k)$ to $\texttt{`map}\langle \texttt{id} \, , \, x \rangle\texttt{`++}\; \texttt{`[]}$ where `id` is the normal form of the identity function on $\texttt{`1} \;\texttt{`}\!\times\; \texttt{`}\alpha_k$.

We leave it to the reader to check that:

$$
\begin{aligned}
\texttt{id} &= \eta\texttt{norm}\; (\texttt{`1} \;\texttt{`}\!\times\; \texttt{`}\alpha_k)\; p \\
\texttt{id} &= \texttt{`}\lambda\, p.\, \eta\texttt{norm}\; (\texttt{`1} \;\texttt{`}\!\times\; \texttt{`}\alpha_k)\; p \\
&= \texttt{`}\lambda\, p.(\eta\texttt{norm}\; \texttt{`1}\; (\texttt{`}\pi_1\, p) \;\texttt{`,}\; \eta\texttt{norm}\; \texttt{`}\alpha_k\; (\texttt{`}\pi_2\, p)) \\
&= \texttt{`}\lambda\, p.(\texttt{`}\langle\rangle \;\texttt{`,}\; \texttt{`}\pi_2\, p)
\end{aligned}
$$

Hence the final standard form of $(\texttt{`}\lambda\, x.x) \,\texttt{`\$}(\texttt{`}\lambda\, x.x)$:

$$
\texttt{`}\lambda\, x.\, \texttt{`map}\langle \texttt{`}\lambda\, p.(\texttt{`}\langle\rangle \;\texttt{`,}\; \texttt{`}\pi_2\, p) \, , \, x \rangle\texttt{`++}\; \texttt{`[]}
$$

The grammar of standard terms explicitly defines a hierarchy between stuck functions: appends are forbidden to appear inside maps and both of them have better not be found sitting in a fold. It is but one way to guarantee the existence of standard forms and future extensions hopefully allowing the programmer to add the ν-rules she fancies holding definitionally will have to make sure –for completeness' sake– that such standard forms exist.

4. Formalization of the Procedure

What we are interested in here is to demonstrate the decidability of the equational theory's extension rather than explaining how to prove termination of a big step semantics in Agda and rely on functional induction to prove the different properties. The reader keen on learning about the latter should refer to James Chapman's thesis [17] where he describes a principled solution to proving termination of big step semantics for various calculi. We, on the other hand, will focus on the former: we opted for a version of the algorithm based, in the tradition of normalization by evaluation, on a model construction which basically collapses the layered stages but is trivially terminating by a structural argument.

Type directed partial evaluation (or normalization by evaluation) is a way to compute the canonical forms by using the evaluation mechanism of the host language whilst exploiting the available type information to retrieve terms from the semantical objects. It was introduced by Berger and Schwichtenberg [12] in order to have an efficient normalization procedure for Minlog. It has since been largely studied in different settings:

Danvy's lecture notes [22] review its foundations and presents its applications as a technique to get rid of static redexes when compiling a program. It also discusses various refinements of the naïve approach such as the introduction of let bindings to preserve a call-by-value semantics or the addition of extra reduction rules[1] to get cleaner code generated. Our ν-rules are somehow reminiscent of this approach.

T. Coquand and Dybjer [19] introduced a glued model recording the partial application of combinators in order to be able to build the reification procedure for a combinatorial logic. In this case the naïve approach is indeed problematic given that the `SK` structure is lost when interpreting the terms in the naïve model and is impossible to get back. This was of great use in the design of a model outside the scope of this paper computing only weak-head normal forms [6].

C. Coquand [18] showed in great details how to implement and prove sound and complete an extension of the usual algorithm to a simply-typed lambda calculus with explicit substitutions. This development guided our correctness proofs.

More recently Abel et al. [2, 3] built extensions able to deal with a variety of type theories. Last but not least Ahman and Staton [4, 5] explained how to treat calculi equipped with algebraic effects which can be seen as an extension of the calculus of Watkins et al. [39] extending judgmental equality with equations for concurrency or Filinski's computational λ-calculus. [25]

Remark We will call $\Gamma \vdash_{nf} \sigma$ the typing derivations restricted to standard values as per the previous section's definitions and $\Gamma \vdash_{ne} \sigma$ the corresponding ones for standard neutrals. Standard list will be silently embedded in standard values: the separation of s and v is an important vestige of the syntactic category l of stuck lists but inlining it in the grammar yields exactly the same set of terms.

Remark Following Agda's color scheme, function names and type constructors will be typeset in blue, constructors will appear in green and variables will be left black.

[1] E.g. $n + 0 \rightsquigarrow n$ in a calculus where $\_ + \_$ is defined by case analysis on the first argument and this expression is therefore stuck.

Model The model is defined by induction on the type using an auxiliary inductive definition parametric in its arguments –which guarantees that the definition is strictly positive therefore meaningful– to give a semantical account of lists. One should remember that the calculus enjoys η-rules for unit, product and arrow types; therefore the semantical counterpart of terms with such types need not be more complex than unit, pairs and actual function spaces.

$$\begin{array}{lll}
\mathcal{M}(\Gamma,\ _ &) : & \mathtt{type}_n \rightarrow \mathtt{Set} \\
\mathcal{M}(\Gamma,\ `\mathtt{1} &) = & \top \\
\mathcal{M}(\Gamma,\ `\alpha_k & ) = & \Gamma \vdash_{ne} `\alpha_k \\
\mathcal{M}(\Gamma,\ \sigma\ `\!\times\ \tau &) = & \mathcal{M}(\Gamma,\sigma) \times \mathcal{M}(\Gamma,\tau) \\
\mathcal{M}(\Gamma,\ \sigma\ `\!\!\rightarrow \tau &) = & \forall \Delta, \Gamma \subseteq \Delta \rightarrow \mathcal{M}(\Delta,\sigma) \rightarrow \mathcal{M}(\Delta,\tau) \\
\mathcal{M}(\Gamma,\ `\mathtt{list}\ \sigma &) = & \mathcal{L}(\Gamma,\sigma,\mathcal{M}(.\ ,\sigma))
\end{array}$$

Standardization may trigger new reductions and we have therefore the obligation to somehow store the computational power of the functions part of stuck maps. This is a bit tricky because the domain type of such functions is nowhere related to the overall type of the expression meaning that no induction hypothesis can be used. Luckily these new computations are only ever provoked by neutral terms: they come from function compositions caused by map or map-fold fusions.

$$\frac{\Gamma : \mathtt{Con}(\mathtt{type}_n) \qquad \sigma : \mathtt{type}_n \qquad \mathtt{M}_\sigma : \mathtt{Con}(\mathtt{type}_n) \rightarrow \mathtt{Set}}{\mathcal{L}(\Gamma,\sigma,\mathtt{M}_\sigma) : \mathtt{Set}}$$

$$\frac{}{`[] : \mathcal{L}(\Gamma,\sigma,\mathtt{M}_\sigma)} \qquad \frac{HD : \mathtt{M}_\sigma(\Gamma) \qquad TL : \mathcal{L}(\Gamma,\sigma,\mathtt{M}_\sigma)}{HD\ `\!::\ TL : \mathcal{L}(\Gamma,\sigma,\mathtt{M}_\sigma)}$$

$$\frac{\begin{array}{c} F : \forall \Delta, \Gamma \subseteq \Delta \rightarrow \Delta \vdash_{ne} \tau \rightarrow \mathtt{M}_\sigma(\Delta) \\ xs : \Gamma \vdash_{ne} `\mathtt{list}\ \tau \qquad YS : \mathcal{L}(\Gamma,\sigma,\mathtt{M}_\sigma) \end{array}}{`\mathtt{map}\langle F\ ,\ xs \rangle\!\!+\!\!+\ YS : \mathcal{L}(\Gamma,\sigma,\mathtt{M}_\sigma)}$$

Remark One should notice the Kripke flavour of the interpretation of function types. It is exactly what is needed to write down a weakening operation thus giving the entire model a Kripke-like structure.

Reify and reflect Mutually defined processes allow normal forms $\Gamma \vdash_{nf} \sigma$ to be extracted from elements of the model $\mathcal{M}(\Gamma,\sigma)$ whilst neutral forms $\Gamma \vdash_{ne} \sigma$ can be turned into elements of the model.

Proof. Both $\downarrow_\sigma : \mathcal{M}(\Gamma,\sigma) \rightarrow \Gamma \vdash_{nf} \sigma$ and $\uparrow_\sigma : \Gamma \vdash_{ne} \sigma \rightarrow \mathcal{M}(\Gamma,\sigma)$ are defined by induction on their type index σ.

Unit, base and product types The unit case is trivial: the reification process returns $`\langle\rangle$ while the reflection one produces the only inhabitant of $\top$. The base type case is solved by the embedding of neutrals into normals on one hand and by the identity function on the other hand. The product case is simply discharged by invoking the induction hypotheses: the reification is the pairing of the reifications of the subterms while the reflection is the reflection of the η-expansion of the stuck term. We can now focus on the more subtle cases.

Arrow type The function case is obtained by η-expansion both at the term level (the normal form will start with a $`\lambda$) and the semantical level (the object will be a function). It is here that the fact that the definitions are mutual is really important.

$$\downarrow_{\sigma\, `\!\rightarrow\, \tau} F \stackrel{\text{def}}{=} `\lambda\, x. \downarrow_\tau F(\_, \uparrow_\sigma x)$$

$$\uparrow_{\sigma\, `\!\rightarrow\, \tau} f \stackrel{\text{def}}{=} \lambda \Delta\ inc\ x. \uparrow_\tau (\mathtt{wk}_{inc}(f)\ `\$ \downarrow_\sigma x)$$

Lists The list case is dealt with by recursion on the semantical list for the reification process and a simple injection for the reflection case. We write $\Downarrow_\sigma$ and $\Uparrow_\sigma$ for the helper functions performing reification and reflection on lists of type $`\mathtt{list}\ \sigma$.

$$\begin{array}{ll}
\Downarrow_\sigma `[] & \stackrel{\text{def}}{=} `[] \\
\Downarrow_\sigma HD\ `\!::\ TL & \stackrel{\text{def}}{=} \downarrow_\sigma HD\ `\!::\ \Downarrow_\sigma TL \\
\Downarrow_\sigma `\mathtt{map}\langle f\ ,\ xs \rangle\!\!+\!\!+\ YS & \stackrel{\text{def}}{=} `\mathtt{map}\langle `\lambda\, x. \downarrow_\sigma f(x)\ ,\ xs \rangle\!\!+\!\!+ \Downarrow_\sigma YS
\end{array}$$

This injection corresponds to applying the identity functor and monoid law. Indeed $\lambda\Delta\_.\uparrow_\sigma$ denotes the identity function and has the appropriate type $\forall\Delta, \Gamma \subseteq \Delta \rightarrow \Delta \vdash_{ne} \sigma \rightarrow \mathcal{M}(\Delta,\sigma)$ to fit in the semantical list mapp constructor.

$$\Uparrow_\sigma xs \stackrel{\text{def}}{=} `\mathtt{map}\langle \lambda\Delta\_.\uparrow_\sigma\ ,\ xs \rangle\!\!+\!\!+\ `[]$$

□

Example of $\eta\nu$-expansions provoked by the reflect / reify functions: for xs a neutral list of type $`\mathtt{list}\ (`\mathtt{1}\ `\!\times `\alpha_k)$, we get an expanded version by drowning it in the model and reifying it back:

$$\Downarrow_{`1\, `\times `\alpha_k} (\Uparrow_{`1\, `\times `\alpha_k} xs) = `\mathtt{map}\langle `\lambda\, p.(`\langle\rangle\ `,\ `\pi_2\, p)\ ,\ xs \rangle\!\!+\!\!+\ `[]$$

This showcases the η-expansion of unit, products and functions as well as the use of the identity laws mentioned during the definition of $\Uparrow_\sigma$.

Proving that every term can be normalized now amounts to proving the existence of an evaluation function producing a term T of the model $\mathcal{M}(\Delta,\sigma)$ given a well-typed term t of the language $\Gamma \vdash \sigma$ and a semantical environment $\mathcal{M}^s(\Delta,\Gamma)$. Indeed the definition of the reflection function $\uparrow_\sigma$ together with the existence of environment weakenings give us the necessary machinery to produce a diagonal semantical environment $\mathcal{M}^s(\Gamma,\Gamma)$ which could then be fed to such an evaluation function.

In order to keep the development tidy and have a more modular proof of correctness, it is wise to give this evaluation function as much structure as possible. This is done through a multitude of helper functions explaining what the semantical counterparts of the usual combinators of the calculus (except for lambda which, integrating a weakening to give the model its Kripke structure, is a bit special) ought to look like.

Theorem 4.1 (Evaluation function)**.** *Given a term in $\Gamma \vdash \sigma$ and a semantical environment in $\mathcal{M}^s(\Delta,\Gamma)$, one can build a semantical object in $\mathcal{M}(\Delta,\sigma)$.*

Proof. A simple induction on the term to be evaluated using the semantical counterparts of the calculus' combinators to assemble semantical objects obtained by induction hypotheses discharges most of the goals. See Figure 7 for the details of the code.

In the lambda case, we have the body of the lambda b in $\Gamma \cdot \sigma \vdash \tau$, an evaluation environment R in $\mathcal{M}^s(\Delta,\Gamma)$ and we are given a context E, a proof inc that $\Delta \subseteq E$ and an object S living in $\mathcal{M}(E,\sigma)$. By combining S and a weakening of R along inc, we get an evaluation environment of type $\mathcal{M}^s(E,\Gamma\cdot\sigma)$ which is just what we needed to conclude by using the $\mathcal{M}(E,\tau)$ delivered by the induction hypothesis on b. □

Remark Unlike traditional normalization by evaluation, reflection and reification are used when defining the interpretation of terms in the model. This is made necessary by the presence of syntactical artifacts (stuck lists) in the mapp constructor. Growing the spine of stuck eliminators calls for the reification of these eliminators' parameters and the reflection of the whole stuck expression to re-inject it in the model.

This kind of patterns also appeared in the glueing construction introduced by Coquand and Dybjer in their account of normalization by evaluation for the simply-typed SK-calculus [19] and can

$$
\begin{array}{ll}
\text{`}[] & \mathcal{M}{+\!\!+}\ ZS = ZS \\
HD\ \text{`}{::}\ TL & \mathcal{M}{+\!\!+}\ ZS = HD\ \text{`}{::}\ (TL\ \mathcal{M}{+\!\!+}\ ZS) \\
\text{`}\mathtt{map}\langle F\ ,\ xs\rangle{+\!\!+}\ YS & \mathcal{M}{+\!\!+}\ ZS = \text{`}\mathtt{map}\langle F\ ,\ xs\rangle{+\!\!+}\ (YS\ \mathcal{M}{+\!\!+}\ ZS)
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{M}\mathtt{map}\ F\ \text{`}[] & = & \text{`}[] \\
\mathcal{M}\mathtt{map}\ F\ (HD\ \text{`}{::}\ TL) & = & F(\_, HD)\ \text{`}{::}\ \mathcal{M}\mathtt{map}\ F\ TL \\
\mathcal{M}\mathtt{map}\ F\ (\text{`}\mathtt{map}\langle G\ ,\ xs\rangle{+\!\!+}\ YS) & = & \text{`}\mathtt{map}\langle F\ \mathcal{M}\circ\ G\ ,\ xs\rangle{+\!\!+}\ \mathcal{M}\mathtt{map}\ F\ YS \\
\multicolumn{3}{l}{\quad \text{where } F\ \mathcal{M}\circ\ G = \lambda E\ inc\ t.F(inc, G(inc, t))}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{M}\mathtt{fold}\ C\ N\ \text{`}[] & = & N \\
\mathcal{M}\mathtt{fold}\ C\ N\ (HD\ \text{`}{::}\ TL) & = & C(\_, HD, \_, \mathcal{M}\mathtt{fold}\ C\ N\ TL) \\
\mathcal{M}\mathtt{fold}_\tau\ C\ N\ (\text{`}\mathtt{map}\langle F\ ,\ xs\rangle{+\!\!+}\ YS) & = & \uparrow_\tau \text{`}\mathtt{fold}(c, n, xs) \\
\multicolumn{3}{l}{\quad \text{where}\quad c\ =\ \text{`}\lambda\, x.\, \text{`}\lambda\, y. \downarrow_\tau C(\_, F(\_, x)), \_, \uparrow_\tau y)} \\
\multicolumn{3}{l}{\qquad\qquad\quad n\ =\ \downarrow_\tau \mathcal{M}\mathtt{fold}\ C\ N\ YS}
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{eval}\ x & R = & R(x) \\
\mathrm{eval}\ (\text{`}\lambda\, x.t) & R = & \lambda E\ inc\ S.\ \mathrm{eval}\ t\ (\mathtt{wk}^s{}_{inc}(R), x \mapsto S) \\
\mathrm{eval}\ (f\ \text{`}\$\, x) & R = & (\mathrm{eval}\ f\ R)(\_, \mathrm{eval}\ x\ R) \\
\mathrm{eval}\ (\text{`}\langle\rangle) & R = & \mathtt{tt} \\
\mathrm{eval}\ (a\ \text{`},\ b) & R = & \mathrm{eval}\ a\ R, \mathrm{eval}\ b\ R \\
\mathrm{eval}\ (\text{`}\pi_1\ t) & R = & \pi_1(\mathrm{eval}\ t\ R) \\
\mathrm{eval}\ (\text{`}\pi_2\ t) & R = & \pi_2(\mathrm{eval}\ t\ R) \\
\mathrm{eval}\ (\text{`}[]) & R = & \text{`}[] \\
\mathrm{eval}\ (hd\ \text{`}{::}\ tl) & R = & (\mathrm{eval}\ hd\ R)\ \text{`}{::}\ (\mathrm{eval}\ tl\ R) \\
\mathrm{eval}\ (xs\ \text{`}{+\!\!+}\ ys) & R = & (\mathrm{eval}\ xs\ R)\ \mathcal{M}{+\!\!+}(\mathrm{eval}\ ys\ R) \\
\mathrm{eval}\ (\text{`}\mathtt{map}(f, xs)) & R = & \mathcal{M}\mathtt{map}(\mathrm{eval}\ f\ R)(\mathrm{eval}\ xs\ R) \\
\mathrm{eval}\ (\text{`}\mathtt{fold}(c, n, xs)) & R = & \mathcal{M}\mathtt{fold}(\mathrm{eval}\ c\ R)(\mathrm{eval}\ n\ R)(\mathrm{eval}\ xs\ R)
\end{array}
$$

Figure 7: Evaluation function and semantical counterparts of list primitives

be observed in other variants of normalization by evaluation deciding more exotic equational theories e.g. having β-reduction but no η-rules for the simply-typed λ-calculus [7].

Remark The only place where type information is needed is when reorganizing neutrals following ν-rules e.g. in the semantical fold. The evaluation function is therefore faithful to the staged evaluation approach. The model is indeed related to the algorithm presented earlier on in section 3.1: we *describe* all the computations eagerly for Agda to see the termination argument but a subtle evaluation strategy applied to the produced code could reclaim the behaviour of the layered approach. It would have to form lambda closures in the arrow case, fire eagerly only the reductions eliminating constructors in the $\mathcal{M}\mathtt{map}$, $\mathcal{M}{+\!\!+}$ and $\mathcal{M}\mathtt{fold}$ helper functions thus postponing the execution of the code corresponding to $\eta\nu$-rules to reification time.

Corollary 4.2. *There is a normalization function* norm *turning terms in* $\Gamma \vdash \sigma$ *into normal forms in* $\Gamma \vdash_{nf} \sigma$.

Proof. Given t a term of type $\Gamma \vdash \sigma$ and $\uparrow^s_{\mathtt{id}}$ the function turning a context Γ into the corresponding diagonal semantical environment $\mathcal{M}^s(\Gamma, \Gamma)$, the normalization procedure is given by the composition of evaluation and reification:

$$\mathrm{norm}\, t \stackrel{\text{def}}{=} \downarrow_\sigma(\mathrm{eval}(t, \uparrow^s_{\mathtt{id}}\Gamma))$$

□

5. Correctness

The typing information provided by the implementation language guarantees that the procedure computes terms in normal forms from its inputs and that they have the same type. This is undoubtedly a good thing to know but does not forbid all the potentially harmful behaviours: the empty list is a type correct normal form for any input of type list but it certainly is not a satisfactory answer with respect to $\beta\delta\iota\eta\nu$-equality. Hence the need for a soundness and a completeness theorem tightening the specification of the procedure.

The meta-theory is an ad-hoc extension of the techniques already well explained by Catarina Coquand [18] in her presentation of a simply-typed lambda calculus with explicit substitutions (but no data). Soundness is achieved through a simple logical relation while completeness needs two mutually defined notions explaining what it means for elements of $\mathcal{M}$ to be semantically equal and to behave uniformly on extensionally equal terms.

The reader should think of these logical relations as specifying requirements for a characterization (being equal, being uniform) to be true of an element at some type. The natural deduction style presentation of these recursive functions should then be quite natural for her: read in a bottom-top fashion, they express that the (dependent) conjunction of the hypotheses – the empty conjunction being $\top$– is the requirement for the goal to hold. Hence leading to a natural interpretation:

$$\frac{A \qquad B \qquad C}{F(t)} \quad \rightsquigarrow \quad F(t) = A \times B \times C$$

5.1 Soundness

Soundness amounts to re-building the propositional part of the reducibility candidate argument [26] which has been erased to get the bare bones model. The logical relation $\mathcal{M}(\Gamma, \sigma) \ni t \between T$ relates a semantical object T in $\mathcal{M}(\Gamma, \sigma)$ and a term t in $\Gamma \vdash \sigma$ which is morally the source of the semantical object.

Logical Relation for Soundness $\mathcal{M}(\Gamma, \sigma) \ni t \between T$ is defined by induction on the type σ plus an appropriate inductive definition for the list case $\mathcal{L}(\Gamma, \sigma, \mathtt{M}_\sigma, \mathtt{M}_\sigma\ \_ \between \_) \ni xs \between XS$. Here are the formation rules of these types.

$$\frac{t : \Gamma \vdash \sigma \qquad T : \mathcal{M}(\Gamma, \sigma)}{\mathcal{M}(\Gamma, \sigma) \ni t \between T : \mathtt{Set}}$$

$$\frac{xs : \Gamma \vdash \text{`}\mathtt{list}\ \sigma \qquad XS : \mathcal{L}(\Gamma, \sigma, \mathtt{M}_\sigma) \qquad \mathtt{M}_\sigma\ \_ \between \_ : \forall \Gamma, \Gamma \vdash \sigma \to \mathtt{M}_\sigma \Gamma \to \mathtt{Set}}{\mathcal{L}(\Gamma, \sigma, \mathtt{M}_\sigma, \mathtt{M}_\sigma\ \_ \between \_) \ni xs \between XS : \mathtt{Set}}$$

Remark It should be no surprise to the now experienced reader that the inductive definition of the logical relation for $\text{`}\mathtt{list}\ \sigma$ is parametrized by $\mathtt{M}_\sigma\ \_ \between \_$, the logical relation for elements of type σ which will be lifted to lists, simply to avoid positivity problems. It is ultimately instantiated with the logical relation taken at type σ.

She will also have noticed that the uses of both $\mathcal{M}$ and $\mathcal{L}$ on the left of $\ni$ are but syntactical artifacts to hint at the connection with the model definition. Hence the different arity in the case of the logical relation for lists.

Unit, base and product types The unit and base type cases are, as expected, the simplest ones and the product case is not very much more exciting:

$$\frac{}{\mathcal{M}(\Gamma, \text{`}\mathtt{1}) \ni t \between T} \qquad \frac{t \rightsquigarrow^*_{\beta\delta\iota\eta\nu} T}{\mathcal{M}(\Gamma, \text{`}\alpha_k) \ni t \between T}$$

$$\frac{t \rightsquigarrow^*_{\beta\delta\iota\eta\nu} a\ \text{`},\ b \qquad \begin{array}{c} a : \Gamma \vdash \sigma \\ \mathcal{M}(\Gamma, \sigma) \ni a \between A \end{array} \qquad \begin{array}{c} b : \Gamma \vdash \tau \\ \mathcal{M}(\Gamma, \sigma) \ni b \between B \end{array}}{\mathcal{M}(\Gamma, \sigma\ \text{`}{\times}\ \tau) \ni t \between A, B}$$

Arrow type Function types on the other hand give rise to a Kripke-like structure in two ways: in addition to the quantification on all possible future context which we need to match the model construction, there is also a quantification on all possible source term reducing to the current one.

$$\frac{\begin{array}{c}\forall \Delta (inc\colon \Gamma \subseteq \Delta)\ x\ X, \mathcal{M}(\Delta, \sigma) \ni x \lightning X \to \\ \forall t, t \leadsto^*_{\beta\delta\iota\eta\nu} \mathtt{wk}_{inc} f \text{ `\$ } x \to \mathcal{M}(\Delta, \tau) \ni t \lightning F(inc, X)\end{array}}{\mathcal{M}(\Gamma, \sigma \text{ `}\!\!\to \tau) \ni f \lightning F}$$

Lists The cases for nil and cons are simply saying that the source term indeed reduces to a term with the corresponding head-constructors and that the eventual subterms are also related to the sub-objects.

$$\frac{t \leadsto^*_{\beta\delta\iota\eta\nu} \text{`[]}}{\mathcal{L}(\Gamma, \sigma, \mathtt{M}_\sigma, \mathtt{M}_\sigma \ \_ \lightning \_) \ni t \lightning \text{`[]}}$$

$$\frac{\begin{array}{c} t \leadsto^*_{\beta\delta\iota\eta\nu} hd \text{ `:: } tl \\ \mathtt{M}_\sigma\ hd \lightning HD \qquad \mathcal{L}(\Gamma, \sigma, \mathtt{M}_\sigma, \mathtt{M}_\sigma \ \_ \lightning \_) \ni tl \lightning TL \end{array}}{\mathcal{L}(\Gamma, \sigma, \mathtt{M}_\sigma, \mathtt{M}_\sigma \ \_ \lightning \_) \ni t \lightning HD \text{ `:: } TL}$$

The mapp case is a bit more complex. The source term is expected to reduce to a term with the same canonical shape and then we expect the semantical function to behave like the one discovered.

$$\frac{\begin{array}{c} t \leadsto^*_{\beta\delta\iota\eta\nu} \text{`}\mathtt{map}(f, xs) \text{ `++ } ys \\ \forall \Delta (inc\colon \Gamma \subseteq \Delta)\ t \to \mathtt{M}_\sigma\ \mathtt{wk}_{inc}(f) \text{ `\$ } t \lightning F(inc, t) \\ \mathcal{L}(\Gamma, \sigma, \mathtt{M}_\sigma, \mathtt{M}_\sigma \ \_ \lightning \_) \ni ys \lightning YS \end{array}}{\mathcal{L}(\Gamma, \sigma, \mathtt{M}_\sigma, \mathtt{M}_\sigma \ \_ \lightning \_) \ni t \lightning \text{`}\mathtt{map}\langle F\ ,\ xs \rangle \text{`++ } YS}$$

The first thing to notice is that whenever two objects are related by this logical relation then the property of interest holds true i.e. the semantical object indeed is a reduct of the source term. This result which mentions the reifying function has to be proven together with the corresponding one about the mutually defined reflection function.

Pointwise extension We denote by $\mathcal{M}^s(\_, \_) \ni \_ \lightning\lightning \_$ the pointwise extension of the soundness logical relation to parallel substitutions and semantical environments.

Lemma 5.1. *Reflect and reify are compatible with this logical relation in the sense that:*

1. *If t_{ne} is a neutral $\Gamma \vdash_{ne} \sigma$ then $\mathcal{M}(\Gamma, \sigma) \ni t_{ne} \lightning \uparrow_\sigma t_{ne}$.*
2. *If t and T are such that $\mathcal{M}(\Gamma, \sigma) \ni t \lightning T$ then $t \leadsto^*_{\beta\delta\iota\eta\nu} \downarrow_\sigma T$*

The Kripke-style structure we mentioned during the definition of the logical relation adds just what is need to have it closed under anti-reductions of the source term:

Proposition 5.2. *For all s and t in $\Gamma \vdash \sigma$, if $s \leadsto^*_{\beta\delta\iota\eta\nu} t$ then for all T such that $\mathcal{M}(\Gamma, \sigma) \ni t \lightning T$, it is also true that $\mathcal{M}(\Gamma, \sigma) \ni s \lightning T$*

The proof of soundness then mainly involves showing that the semantical counterparts of the language's combinators we defined during the model construction are compatible with the logical relation. Namely that e.g. if $\mathcal{M}(\Gamma, \sigma \text{ `}\!\!\to \tau) \ni f \lightning F$ and $\mathcal{M}(\Gamma, \text{`}\mathtt{list}\ \sigma) \ni xs \lightning XS$ hold then it is also true that: $\mathcal{M}(\Gamma, \text{`}\mathtt{list}\ \tau) \ni \text{`}\mathtt{map}(f, xs) \lightning \text{`}\mathtt{map}(F, XS)$.

Theorem 5.3. *Given a term $t\colon \Gamma \vdash \sigma$, a parallel substitution $\rho\colon \Delta \vDash^s \Gamma$ and an evaluation environment R such that ρ and R are related ($\mathcal{M}^s(\Delta, \Gamma) \ni \rho \lightning\lightning R$ holds), the evaluation of t in R is related to $t[\rho]$: $\mathcal{M}(\Delta, \sigma) \ni t[\rho] \lightning \mathrm{eval}(t, R)$*

Proof. The theorem is proved by structural induction on the shape of the typing derivation of t. The variable case is trivially discharged by using the proof of $\mathcal{M}^s(\Delta, \Gamma) \ni \rho \lightning\lightning R$.

All the other cases – except for the lambda one – can be solved by combining induction hypotheses with the appropriate lemma proving that the corresponding semantical combinator respects the logical relation.

In the case where $t = \text{`}\lambda\, x.b$, we are given a context $E$ together with a proof $inc$ that it is an extension of $\Delta$, a term $u$ and an object $U$ which are related $\mathcal{M}(E, \sigma) \ni u \lightning U$ and, finally, a term $s\colon E \vdash \tau$ which reduces to $(\text{`}\lambda\, x.b)[\rho] \text{ `\$ } u$. First of all, we should notice that $s \leadsto^*_{\beta\delta\iota\eta\nu} b[\rho, x \mapsto u]$ and therefore that to prove $\mathcal{M}(E, \tau) \ni s \lightning T$ it is enough to prove that $\mathcal{M}(E, \tau) \ni b[\rho, x \mapsto u] \lightning T$. And we get just that by using the induction hypothesis with the related parallel substitution ρ' and evaluation environment R' obtained by the combination of the weakening of ρ (resp. R) along inc with u (resp. U). □

Corollary 5.4. *A term t reduces to the normal form produced by the normalization by evaluation procedure: $t \leadsto^*_{\beta\delta\iota\eta\nu} \mathrm{norm}\, t$. And if two terms t and u have the same normal form up-to α-equivalence then they are indeed related: $t \equiv_{\beta\delta\iota\eta\nu} u$.*

Proof. The identity parallel substitution is related to the diagonal evaluation environment and $t[\mathtt{id}_\Gamma]$ is equal to t hence, by the previous theorem, $\mathcal{M}(\Gamma, \sigma) \ni t \lightning \mathrm{eval}(t, \mathtt{id}_{\mathcal{M}^s\ \Gamma})$ and then $t \leadsto^*_{\beta\delta\iota\eta\nu} \mathrm{norm}\, t$. □

5.2 Completeness

Completeness can be summed up by the fact that the interpretation of $\beta\delta\iota\eta\nu$-convertible elements produces semantical objects behaving similarly. This notion of similar behaviour is formalized as *semantic equality* where, in the function case, we expect both sides to agree on any *uniform* input rather than any element of the model. As usual the list case is dealt with by using an auxiliary definition parametric in its "interesting" arguments.

Definition The semantic equality of two elements T, U of $\mathcal{M}(\Gamma, \sigma)$ is written $T \equiv_\sigma U$ while $T \in \mathcal{M}(\Gamma, \sigma)$ being uniform is written $\mathtt{Uni}_\sigma\ T$. They are both mutually defined by induction on the index σ in Figure 8.

Quite unsurprisingly, the unit case is of no interest: all the semantical units are equivalent and uniform. Semantic equality for elements with base types is up-to α-equivalence: inhabitants are just bits of data (neutrals) which can be compared in a purely syntactical fashion because we use nameless terms. They are always uniform.

In the product case, the semantical objects are actual pairs and the definition just forces the properties to hold for each one of the pair's components.

The function type case is a bit more hairy. While extensionality on uniform arguments is simple to state, uniformity has to enforce a lot of invariants: application of uniform objects should yield a uniform object, application of extensionally equal uniform objects should yield extensionally equal objects and weakening and application should commute (up to extensionality).

In the $\text{`}\mathtt{list}\ \sigma$ case, extensional equality is an inductive set basically building the (semantical) diagonal relation on lists of the same type. It is parametrized by a relation EQ_σ on terms of type $\mathcal{M}(\Delta, \sigma)$ (for any context Δ) which is, in the practical case instantiated with $\_ \equiv_\sigma \_$ as one would expect. Uniformity is, on the other hand, defined by recursion on the semantical list. It could very well be defined as being parametric in something behaving like $\mathtt{Uni}_\sigma\ \_$ but this is not necessary: there are no positivity problems! It is therefore probably better to stick to a lighter presentation here. The empty list indeed is uniform. A constructor-headed list is said to be uniform if its head of type $\mathcal{M}(\Gamma, \sigma)$ is uniform and its tail also is uniform. The criterion for a stuck list is a bit more involved.

$$\frac{}{T \equiv_{`1} U} \qquad \frac{}{\mathtt{Uni}_{`1}\ T} \qquad \frac{T = U}{T \equiv_{`\alpha_k} U} \qquad \frac{}{\mathtt{Uni}_{`\alpha_k}\ T} \qquad \frac{A \equiv_\sigma C \qquad B \equiv_\tau D}{(A,B) \equiv_{\sigma\ `\times\ \tau} (C,D)} \qquad \frac{\mathtt{Uni}_\sigma\ A \qquad \mathtt{Uni}_\tau\ B}{\mathtt{Uni}_{\sigma\ `\times\ \tau}\ (A,B)}$$

$$\frac{\forall\Delta(inc : \Gamma \subseteq \Delta)(S : \mathcal{M}(\Delta,\sigma)) \to \mathtt{Uni}_\sigma\ S \to F(inc,S) \equiv_\tau G(inc,S)}{F \equiv_{\sigma\ `\hookrightarrow\ \tau} G}$$

$$\frac{\begin{array}{c}\forall\Delta(inc : \Gamma \subseteq \Delta), \mathtt{Uni}_\sigma\ S \to \mathtt{Uni}_\tau\ F(inc,S) \\ \forall\Delta(inc : \Gamma \subseteq \Delta) \to \mathtt{Uni}_\sigma\ S_1 \to \mathtt{Uni}_\sigma\ S_2 \to S_1 \equiv_\sigma S_2 \to F(inc,S_1) \equiv_\tau F(inc,S_2) \\ \forall inc_1, inc_2 \to \mathtt{Uni}_\sigma\ S \to \mathtt{wk}_{inc_1}\ F(inc_2,S) \equiv_\tau F(inc_2 \cdot inc_1, \mathtt{wk}_{inc_1}\ S)\end{array}}{\mathtt{Uni}_{\sigma\ `\hookrightarrow\ \tau}\ F}$$

$$\frac{}{`[] : `[] \equiv_\sigma^{`\mathtt{list}} `[]} \qquad \frac{}{\mathtt{Uni}_{`\mathtt{list}\ \sigma}\ `[]} \qquad \frac{hd : EQ_\sigma(X,Y) \qquad tl : XS \equiv_\sigma^{`\mathtt{list}} YS}{hd\ `{::}\ tl : X\ `{::}\ XS \equiv_\sigma^{`\mathtt{list}} Y\ `{::}\ YS} \qquad \frac{\mathtt{Uni}_\sigma\ HD \qquad \mathtt{Uni}_{`\mathtt{list}\ \sigma}\ TL}{\mathtt{Uni}_{`\mathtt{list}\ \sigma}\ HD\ `{::}\ TL}$$

$$\frac{xs : xs_1 \equiv xs_2 \qquad YS : YS_1 \equiv_\sigma^{`\mathtt{list}} YS_2 \qquad F : \forall\Delta(inc : \Gamma \subseteq \Delta)(t : \Delta \vdash_{ne} \tau), EQ_\sigma(F_1(inc,t), F_2(inc,t))}{`\mathtt{map}(F,xs)\ `{+\!\!+}\ YS : \qquad `\mathtt{map}(F_1,xs_1)\ `{+\!\!+}\ YS_1 \equiv_\sigma^{`\mathtt{list}} `\mathtt{map}(F_2,xs_2)\ `{+\!\!+}\ YS_2}$$

$$\frac{\forall\Delta(inc : \Gamma \subseteq \Delta)(t : \Delta \vdash_{ne} \tau), \mathtt{Uni}_\sigma\ F(inc,t) \qquad \forall inc_1, inc_2, t, \mathtt{wk}_{inc_1}\ F(inc_2,t) \equiv_\sigma F(inc_2 \cdot inc_1, \mathtt{wk}_{inc_1}\ t) \qquad \mathtt{Uni}_{`\mathtt{list}\ \sigma}\ YS}{\mathtt{Uni}_{`\mathtt{list}\ \sigma}\ `\mathtt{map}(F,xs)\ `{+\!\!+}\ YS}$$

Figure 8: Semantic equality and uniformity of objects in the model

Mimicking the definition of uniformity for functions, there are two requirements on the stuck map: applying it to a neutral yields a uniform element of the model and application and weakening commute. Lastly the second argument of the stuck append should be uniform too.

Remark The careful reader will already have noticed that this defines a family of equivalence relations; we will not make explicit use of reflexivity, symmetry and transitivity in the paper but it is fundamental in the formalization.

Recall that the completeness theorem was presented as expressing the fact that elements equivalent with respect to the reduction relation were interpreted as semantical objects behaving similarly. For this approach to make sense, knowing that two semantical objects are extensionally equal should immediately imply that their respective reifications are syntactically equal. Which is the case.

Lemma 5.5. *Reification, reflection and weakenings are compatible with the notions of extensional equality and uniformity.*

1. *If $T \equiv_\sigma U$ then $\downarrow_\sigma T = \downarrow_\sigma U$*
2. *If t_{ne} is a neutral $\Gamma \vdash_{ne} \sigma$ then $\mathtt{Uni}_\sigma\ (\uparrow_\sigma t_{ne})$*
3. *Weakening and reification commute for uniform objects*

Now that we know that all the theorem proving ahead of us will not be meaningless, we can start actually tackling completeness. When applying an extensional function, it is always required to prove that the argument is uniform. Being able to certify the uniformity of the evaluation of a term is therefore of the utmost importance.

Lemma 5.6. *Evaluation preserves properties of the evaluation environment.*

1. *Evaluation in uniform environments produces uniform values*
2. *Evaluation in semantically equivalent environments produces semantically equivalent values*
3. *Weakening the evaluation of a term is equivalent to evaluating this term in a weakened environment*

Theorem 5.7. *If s and t are two terms in $\Gamma \vdash \sigma$ such that $s \leadsto_{\beta\delta\iota\eta\nu} t$ and if R is a uniform environment in $\mathcal{M}^s(\Delta,\Gamma)$ then* $\mathrm{eval}(s,R) \equiv_\sigma \mathrm{eval}(t,R)$.

Proof. One proceeds by induction on the proof that s reduces to t.

Structural rules Structural rules can be discharged by combining induction hypotheses and reflexivity proofs using previously proved lemma such as the fact that evaluation in uniform environments yields uniform elements for the structural rule for the argument part of application.

$\beta\iota$-rules Each one the ι rules holds by reflexivity of the extensional equality, indeed evaluation realizes these computation rules syntactically. The case of the β rule is slightly more complicated. Given a function $`\lambda\, x.b$ and its argument u, one starts by proving that the diagonal semantical environment extended with the evaluation of u in R is extensionally equal to the evaluation in R of the diagonal substitution extended with u. Thence, knowing that the evaluations of a term in two extensionally equal environments are extensionally equal, one can see that the evaluation of the redex is related to the evaluation of the body in an environment corresponding to the evaluation of the substitution generated when firing the redex. Finally, the fact that eval and substitution commute (up-to-extensionality) lets us conclude.

$\eta\nu$-rules definitely are the most work-intensive ones: except for the ones for product and unit types which can be discharged by reflexivity of the semantic equality, all of them need at least a little bit of theorem proving to go through. It is possible to prove the map-id, map-append, append-nil, associativity of append and various fusion rules by induction on the 'nut' for uniform values. Solving the goals is then just a matter of combining the right auxiliary lemma with facts proved earlier on, typically the uniformity of semantical object obtained by evaluating a term in a uniform environment. □

Corollary 5.8 (Completeness)**.** *For all terms t and u of type $\Gamma \vdash \sigma$, if $t \equiv_{\beta\delta\iota\eta\nu} u$ then* $\mathrm{norm}\ t = \mathrm{norm}\ u$.

Proof. Reflection produces uniform values and uniformity is preserved through weakening hence the fact that the trivial diagonal environment is uniform. Combined with iterations of the previous lemma along the proof that $t \equiv_{\beta\delta\iota\eta\nu} u$, we get that the respective evaluations of t and u are extensionally equal which we have proved to be enough to get syntactically equal reifications. □

Corollary 5.9. *The equational theory enriched with ν-rules is decidable.*

Proof. Given terms t and u of the same type $\Gamma \vdash \sigma$, we can get two normal forms $t_{nf} = \mathrm{norm}\, t$ and $u_{nf} = \mathrm{norm}\, u$ and test them for equality up-to α-conversion (which is a simple syntactic check in our nameless representation in Agda).

If $t_{nf} = u_{nf}$ then the soundness result allows us to conclude that t and u are convertible terms.

If $t_{nf} \neq u_{nf}$ then t and u are not convertible. Indeed, if they were then the completeness result guarantees us that t_{nf} and u_{nf} would be equal which leads to a contradiction. □

Example of terms which are identified thanks to the internalization of the ν-rules.

1. In a context with two functions f and g of type σ ‘$\to$ ‘`1`, ‘$\lambda\, xs.$ ‘`map`(f, xs) and ‘$\lambda\, xs.$ ‘`map`(g, xs) both normalize to ‘$\lambda\, xs.$ ‘`map`(‘$\lambda\, \_.$ ‘$\langle\rangle, xs)$ ‘$+\!\!+$ ‘$[]$ and are therefore declared equal.
2. At type $\Gamma \vdash$ ‘`list` (‘α_k ‘$\times$ ‘α_l) ‘$\to$ ‘`list` (‘α_k ‘$\times$ ‘α_l), the terms ‘$\lambda\, xs.xs$ and ‘$\lambda\, xs.$ ‘`map`$(swap,$ ‘`map`$(swap, xs))$ where $swap$ is the function ‘$\lambda\, p.$(‘$\pi_2\, p$ ‘, ‘$\pi_1\, p$) swapping the order of a pair's elements are convertible with normal form ‘$\lambda\, xs.$ ‘`map`(‘$\lambda\, p.$(‘$\pi_1\, p$ ‘, ‘$\pi_2\, p$)$, xs)$ ‘$+\!\!+$ ‘$[]$.

6. Scaling up to Type Theory

Now that we know for sure that the judgmental equality can be safely extended with some ν-rules, we are ready to tackle more complex type theories. We have already experimented with extending our simply-typed setting to a universe of polynomial datatypes with map and fold. We have to identify which parts of the setting are key to the success of this technique and how to enforce that the generalized version still has good properties.

Types Arrow types will be replaced by Π-types and product types by Σ-types but the basic machinery of evaluation and type-directed η-expansion work in much the same way.

In Type Theory, it is not quite enough to be able to decide the judgmental equality. Pollack's PhD thesis ([37], Section 5.3.1), taught us how to turn the typing relation with a conversion rule into a syntax-directed typechecking algorithm by relying on ordinary evaluation (cf. the application typing rule in Figure 9). It is therefore quite crucial for ensuring the reusability of previous typechecking algorithms to be able to guarantee that ordinary evaluation is complete for uncovering constructor-headed terms i.e. $\Gamma \vdash t \equiv C\ \vec{t_i} : T$ should imply that $t \rightsquigarrow^\star C\ \vec{t_i}'$. This can be enforced by making sure that candidates for ν-rules are only reorganizing spines of stuck eliminators and are absolutely never emitting new constructors.

$$\frac{\Gamma \vdash f : F \qquad F \rightsquigarrow^\star (x : S) \to T \qquad \Gamma \vdash s : S' \qquad \Gamma \vdash S \equiv S' : \mathtt{Set}}{\Gamma \vdash f\, s : T[s/x]}$$

Figure 9: Syntax-directed typing rule for application, Pollack [37]

η-rules A Type Theory does not need to have judgmental η-rules for the ν-rules to make sense. However this partially defeats the purpose of this extension: without η-rules for products we fail to identify the silly identity on lists of products `map swap . map swap` with the more traditional one $\lambda x.x$ because $f_1 =$ ‘$\lambda\, x.x$ is different from $f_2 =$ ‘$\lambda\, x.$(‘$\pi_1\, x$ ‘, ‘$\pi_2\, x$) when both terms would reduce respectively to ‘$\lambda\, x.$ ‘`map`$\langle f_1 , x\rangle$‘$+\!\!+$ ‘$[]$ and ‘$\lambda\, x.$ ‘`map`$\langle f_2 , x\rangle$‘$+\!\!+$ ‘$[]$. So close yet so far away!

Defined symbols In this presentation, a handful of functions are built-in rather than user-defined. This will probably be one of the biggest changes when moving to a usable Type Theory. We can enforce that functions defined by pattern-matching have a fixed arity and are always fully applied at that arity. Such a function is stuck if it is strict in a neutral argument. Some type theories reduce pattern matching to the primitive elimination operator for each datatype. To apply ν-rules, we need to detect which stuck eliminators correspond to which stuck pattern matches. This is the same problem as producing readable output from normalizing open terms, and it has already been solved by the 'labelled type' translation used in Epigram, which effectively inserts documentation of stuck pattern matches into spines of stuck eliminators [34].

Criteria for ν-rules Working in a setting where the datatypes are given by a universe [16], we should at least expect that built-in generic operators, e.g. map, have associated ν-rules. However, it is clearly desirable to allow the programmer to propose ν-rules for programs of her own construction. How will the machine check that proposed ν-rules keep evaluation canonical and judgmental equality consistent and decidable? We have already seen that ν-rules must avoid to emit new constructors; this can be summed up by the *mantra*: "A ν-rule may restart computation *within* its contractum but *never* in its enclosing context".

The candidates for ν-rules should hold trivially by a Boyer-Moore style induction; in other words, the $\beta\delta\iota - \nu$ critical pairs should be convergent. This tells us that these rules are consistent and can be delayed until after evaluation.

Obviously, the $\nu - \nu$ critical pairs should also be convergent. These three criteria are all easy to check provided that ν-reductions give rise to a terminating term rewrite system.

This termination requirement is the last criterion. As a first instance, a rather conservative approach could be to ask the user for a linear order on defined symbols which we would lift to expressions by using the lexicographic ordering of the encountered defined symbols starting from the "nut" and going outwards. If this ordering is compatible with a left to right orientation of the ν-rules she wants to hold, then it is terminating. In the set of ν-rules used as an example in this paper, the simple ordering ‘$+\!\!+$ $>$ ‘`map` $>$ ‘`fold` is compatible with the rules.

7. Further Opportunities for ν-Rules

We were motivated to develop a proof technique for extending definitional equality with ν-rules because there are many opportunities where we might profit by doing so. Let us set out a prospectus.

Reflexive coercion for type-based equality. Altenkirch, McBride and W. Swierstra developed a propositional equality for intensional type theory [9] which differs from the usual inductive definition (`refl a : a = a`) in that its main eliminator

$$\frac{S, T : \mathtt{Set} \quad Q : S = T \quad s : S}{s[Q : S = T\rangle : T}$$

computes by structural recursion first on the *types* S and T, and then (where appropriate) on s, rather than by pattern matching on the proof Q. Equality is still reflexive, so evaluation can leave us with terms $n[\mathtt{refl}\ n : N = N\rangle : N$ where n is a neutral term in a neutral type N. It is clearly a nuisance that this term does not compute to n, as would happen if the eliminator matched on the proof. The fix is to add a ν-rule which discards coercions whenever it is type-safe to do so:

$$\boxed{\boxed{s}[Q : S = T\rangle} = \boxed{s} \qquad \text{if } S \equiv T : \mathtt{Set}$$

It is easy to check that adding this rule for neutral terms makes it admissible for all terms, and hence that we need add it not to evaluation, but only to the reification process which follows, just as with the ν-rules in this paper. There, as here, this spares the evaluation process from decisions which involve η-expansion and thus require a name supply. The ν-rule thus gives us a non-disruptive means to respect the full computational behaviour of inductive equality in the observational setting.

Functor laws. Barral and Soloviev give a treatment of functor laws for parametrized inductive datatypes by modifying the ι-rules of their underlying type theory [11]. We should very much hope to achieve the same result, as we did here in the special case of lists, just by adding ν-rules. Our preliminary experiments [33] suggest that we can implement functor laws once and for all in a type theory whose datatypes are given once and for all by a syntactic encoding of strictly positive functors, as Dagand and colleagues propose [16, 20]. Moreover, Luo and Adams have shown [31] that structural subtyping for inductive types can be reified by a coherent system of implicit coercions if functor laws hold definitionally.

Monad laws. Watkins et al. give a definitional treatment of monad laws in order to achieve an adequate representation of concurrent processes encapsulated monadically in a logical framework [39]. For straightforward free monads, an experimental extension of Epigram (by Norell, as it happens) [33] suggests that we may readily allow ν-rules:

$$\boxed{\boxed{t} \gg= \mathtt{return}} = \boxed{t} \qquad \boxed{(\boxed{t} \gg= \sigma) \gg= \rho} = \boxed{\boxed{t} \gg= ((\gg= \sigma) \cdot \rho)}$$

Atkey's Foveran system uses a similar normalization method for free monad laws [10], again for an encoded universe of underlying functors.

Decomposing functors. Dagand and colleagues further note that their syntax of descriptions for indexed functors is, by virtue of being a syntax, itself presentable as the free monad of a functor. The description decoder

$$\mathtt{Decode} : \mathtt{IDesc}\ I \to (I \to \mathtt{Set}) \to \mathtt{Set}$$

is structurally recursive in the description and lifts pointwise to an interpretation of substitutions in the $\mathtt{IDesc}$ monad

$$\begin{array}{l} \llbracket \_ \rrbracket : (O \to \mathtt{IDesc}\ I) \quad \to \quad (I \to \mathtt{Set}) \to (O \to \mathtt{Set}) \\ \llbracket \sigma \rrbracket\ X\ o = \mathtt{Decode}\ (\sigma\ o)\ X \end{array}$$

as indexed functors with a 'map' operation satisfying functor laws. However, not only does this interpretation *deliver* functors, it is *itself* a contravariant functor: the identity substitution yields the identity functor just by $\beta\delta\iota$, but we may also interpret Kleisli composition as reverse functor composition

$$\llbracket (\gg= \sigma) \cdot \rho \rrbracket = \llbracket \rho \rrbracket \cdot \llbracket \sigma \rrbracket$$

by means of a ν-rule

$$\boxed{\mathtt{Decode}\ \boxed{(\boxed{D} \gg= \sigma)}\ \ X} = \boxed{\mathtt{Decode}\ \ \boxed{D}\ \ (\llbracket \sigma \rrbracket\ X)}$$

taking each D to be some $\rho\ o$. If we want to do a 'scrap your boilerplate' style traversal of some described container-like structure, we need merely exhibit the decomposition of the description as some $(\gg= \sigma) \cdot \rho$, where ρ describes the invariant superstructures and σ the modified substructures, then invoke the functoriality of $\llbracket \rho \rrbracket$. This ν-rule thus lets us expose functoriality over substructures not anticipated by explicit parametrization in datatype declarations. We thus recover the kind of ad-hoc data traversal popularized by Lämmel and Peyton Jones [30] by static structural means.

Universe embeddings. A type theory with inductive-recursive definitions is powerful enough to encode universes of dependent types by giving a datatype of codes *in tandem* with their interpretations [24], the paradigmatic example being

$$\begin{array}{l|l} \mathtt{U}_1 : \mathtt{Set} & \mathtt{El}_1 : \mathtt{U}_1 \to \mathtt{Set} \\ \text{`}\mathtt{Pi}_1 : (S : \mathtt{U}_1) \to & \mathtt{El}_1\ (\text{`}\mathtt{Pi}_1\ S\ T) = \\ \qquad\qquad (\mathtt{El}_1\ S \to \mathtt{U}_1) \to \mathtt{U}_1 & \quad (s : \mathtt{El}_1\ S) \to \mathtt{El}_1\ (T\ s) \\ \vdots & \vdots \end{array}$$

Palmgren [36] suggests that one way to model a cumulative hierarchy of such universes is to give each a code in the next, so

$$\begin{array}{l|l} \mathtt{U}_2 : \mathtt{Set} & \mathtt{El}_2 : \mathtt{U}_2 \to \mathtt{Set} \\ \text{`}\mathtt{U}_1 : \mathtt{U}_2 & \mathtt{El}_2\ \text{`}\mathtt{U}_1 = \mathtt{U}_1 \\ \text{`}\mathtt{Pi}_2 : (S : \mathtt{U}_2) \to & \mathtt{El}_2\ (\text{`}\mathtt{Pi}_2\ S\ T) = \\ \qquad\qquad (\mathtt{El}_2\ S \to \mathtt{U}_2) \to \mathtt{U}_2 & \quad (s : \mathtt{El}_2\ S) \to \mathtt{El}_2\ (T\ s) \\ \vdots & \vdots \end{array}$$

and then define an embedding recursively

$$\begin{array}{l} \uparrow : \mathtt{U}_1 \to \mathtt{U}_2 \\ \uparrow (\text{`}\mathtt{Pi}_1\ S\ T) = \text{`}\mathtt{Pi}_2\ (\uparrow S)\ (\lambda s.\ \uparrow (T\ s)) \end{array}$$

but a small frustration with this proposal is that s is abstracted at type $\mathtt{El}_2\ (\uparrow S))$, but used at type $\mathtt{El}_1\ S$, and these two types are not definitionally equal for an abstract S. One workaround is to make $\uparrow$ a constructor of $\mathtt{U}_2$, at the cost of some redundancy of representation, but now we might also consider fixing the discrepancy with a ν-rule

$$\boxed{\mathtt{El}_2\ \boxed{(\uparrow \boxed{S})}} = \boxed{\mathtt{El}_1\ \boxed{S}}$$

This is peculiar for our examples thus far, in that the ν-rule is needed even to typecheck the $\delta\iota$-rules for $\uparrow$, reflecting the fact that $\uparrow$ should not be any old function from $\mathtt{U}_1$ to $\mathtt{U}_2$, but rather one which preserves the meanings given by $\mathtt{El}_1$ and $\mathtt{El}_2$. In effect, the ν-rule is expressing the coherence property of a richer notion of morphism. It is inviting to wonder what other notions of coherence we might enable and enforce by checking that ν-rules hold of the operations we implement.

Non-examples. A key characteristic of a ν-rule is that it is a nut-preserving rearrangement of neutral term layers. Whilst this is good for associativity and sometimes for distributivity, it is perfectly useless for commutativity. Suppose $+$ for natural numbers is recursive on its first argument, and observe that rewriting $x + y$ to $y + x$ when x is neutral will not result in a neutral term unless y is also neutral. Less ambitious rules such as $x + \mathtt{suc}\ y = \mathtt{suc}\ (x + y)$ and $x * 0 = 0$ similarly make neutral terms come unstuck, and so cannot be postponed until reification if we want to be sure that evaluation suffices to show whether any expression in a datatype can be put into constructor-headed form. Walukiewicz-Chrzaszcz has proposed a more invasive adoption of rewriting for Coq, necessitating a modified evaluator, but incorporating rules which can expose constructors [38]. Her untyped rewriting approach sits awkwardly with η-laws, but we can find a more carefully structured compromise.

8. Discussion

We fully expect to scale this technology up to type theory. Abel and Dybjer (with Aehlig [2] and T. Coquand [3]) have already given normalization by evaluation algorithms which we plan to adapt.

Finding good criteria for checking that candidate ν-rules can safely be added is of the utmost importance. We want to let the programmer negotiate the new ν-rules she wants, as long as the machine can check that they yield a notion of standard form and lift from neutral terms to all terms by the prior equational theory.

It is also interesting to try to integrate ν-rules with more practical presentations of normalization. For instance Grégoire and Leroy's conversion by compilation to a bytecode machine derived from Ocaml's ZAM [27] decides η by expansion only when provoked by a λ: such laziness is desirable when possible but causes trouble with η-rules for unit types and may conceal the potential to apply ν-rules. Hereditary substitution [39], formalized by Abel [1] and by Keller and Altenkirch [29], may be easier to adapt.

Acknowledgements

We would like to thank the anonymous reviewers for their helpful comments and suggestions as well as Stevan Andjelkovic for carefully reading our draft.

References

[1] A. Abel. Implementing a normalizer using sized heterogeneous types. *Journal of Functional Programming*, 19(3-4):287–310, 2009.

[2] A. Abel, K. Aehlig, and P. Dybjer. Normalization by evaluation for Martin-Löf type theory with one universe. *Electr. Notes Theor. Comput. Sci*, 173:17–39, 2007.

[3] A. Abel, T. Coquand, and P. Dybjer. Normalization by evaluation for Martin-Löf type theory with typed equality judgements. In *LICS*, pages 3–12. IEEE Computer Society, 2007.

[4] D. Ahman. Computational effects, algebraic theories and normalization by evaluation. Master's thesis, University of Cambridge, June 2012.

[5] D. Ahman and S. Sam. Normalization by evaluation and algebraic effects. In *29th Conference on Mathematical Foundations of Programming Semantics, MFPS XXIX, New Orleans, LA, June 2013*, 2013.

[6] G. Allais. Forge crowbars, Acquire normal forms. Technical report, University of Strathclyde, 2012. URL `http://gallais.org/pdf/report2012.pdf`.

[7] G. Allais. Glueing terms to models - variations on nbe. Slides, May 2013. URL `http://gallais.org/pdf/awayday13.pdf`.

[8] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *CSL*, pages 453–468, 1999.

[9] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In A. Stump and H. Xi, editors, *PLPV*, pages 57–68. ACM, 2007.

[10] R. Atkey. A type checker that knows its monad from its elbow. Blog post, December 2011. URL `http://bentnib.org/posts/2011-12-14-type-checker.html`.

[11] F. Barral and S. Soloviev. Inductive type schemas as functors. In D. Grigoriev, J. Harrison, and E. A. Hirsch, editors, *CSR*, volume 3967 of *Lecture Notes in Computer Science*, pages 35–45. Springer, 2006.

[12] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *LICS: IEEE Symposium on Logic in Computer Science*, 1991.

[13] P. Boutillier. Equality for lambda-terms with list primitives. Internship report, University of Strathclyde, ENS Lyon, July 2009.

[14] R. S. Boyer and J. S. Moore. Proving theorems about LISP functions. *J. ACM*, 22(1):129–144, Jan. 1975.

[15] J. Chapman, T. Altenkirch, and C. McBride. Epigram reloaded: a standalone typechecker for ETT. In M. C. J. D. van Eekelen, editor, *Trends in Functional Programming*, volume 6 of *Trends in Functional Programming*, pages 79–94. Intellect, 2005.

[16] J. Chapman, P.-E. Dagand, C. McBride, and P. Morris. The gentle art of levitation. *SIGPLAN Not.*, 45(9):3–14, Sept. 2010.

[17] J. M. Chapman. *Type checking and normalisation*. PhD thesis, University of Nottingham, 2009.

[18] C. Coquand. A formalised proof of the soundness and completeness of a simply typed lambda-calculus with explicit substitutions. *Higher-Order and Symbolic Computation*, 15(1):57–90, 2002.

[19] T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical. Structures in Comp. Sci.*, 7:75–94, Feb. 1997. ISSN 0960-1295.

[20] P.-E. Dagand and C. McBride. Elaborating Inductive Definitions. *ArXiv e-prints*, Oct. 2012.

[21] N. A. Danielsson. A formalisation of a dependently typed language as an Inductive-Recursive family. In T. Altenkirch and C. McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer Berlin Heidelberg, 2007.

[22] O. Danvy. Type-directed partial evaluation. In J. Hatcliff, T. Mogensen, and P. Thiemann, editors, *Partial Evaluation*, volume 1706 of *Lecture Notes in Computer Science*, chapter 16, pages 367–411. Springer Berlin / Heidelberg, Berlin, Heidelberg, 1999.

[23] N. G. de Bruijn. Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation. *Indagationes Mathematicæ*, 34:381–392, 1972.

[24] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In J.-Y. Girard, editor, *TLCA*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer, 1999.

[25] A. Filinski. Normalization by evaluation for the computational Lambda-Calculus. In S. Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, chapter 15, pages 151–165. Springer Berlin / Heidelberg, Berlin, Heidelberg, Apr. 2001.

[26] J. Girard. *Le Point Aveugle: Cours de logique: Tome 1, Vers la perfection*. Hermann, 2006.

[27] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. *SIGPLAN Not.*, 37(9):235–246, 2002.

[28] INRIA. The Coq proof assistant.

[29] C. Keller and T. Altenkirch. Hereditary substitutions for simple types, formalized. In *Proceedings of the third ACM SIGPLAN workshop on Mathematically structured functional programming*, MSFP '10, pages 3–10, New York, NY, USA, 2010. ACM.

[30] R. Lämmel and S. L. P. Jones. Scrap your boilerplate: a practical design pattern for generic programming. In Z. Shao and P. Lee, editors, *TLDI*, pages 26–37. ACM, 2003. ISBN 1-58113-649-8.

[31] Z. Luo and R. Adams. Structural subtyping for inductive types with functorial equality rules. *Mathematical Structures in Computer Science*, 18(5):931–972, 2008.

[32] C. McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In *Proceedings of the 6th ACM SIGPLAN workshop on Generic programming*, WGP '10, New York, NY, USA, 2010. ACM.

[33] C. McBride. EE-PigWeek 4-9 Jan 2010. Blog post, January 2010. URL `http://www.e-pig.org/epilogue/?p=306`.

[34] C. McBride and J. McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, Jan. 2004.

[35] U. Norell. Dependently typed programming in Agda. In P. W. M. Koopman, R. Plasmeijer, and S. D. Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008.

[36] E. Palmgren. On Universes in Type Theory. In G. Sambin and J. Smith, editors, *Twenty Five Years of Constructive Type Theory*. Oxford Univ. Press, 1998.

[37] R. Pollack. *The theory of LEGO – a proof checker for the extendend calculus of constructions*. PhD thesis, University of Edinburgh, 1994.

[38] D. Walukiewicz-Chrzaszcz. Termination of rewriting in the calculus of constructions. *J. Funct. Program.*, 13(2):339–414, 2003.

[39] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. In S. Berardi, M. Coppo, and F. Damiani, editors, *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 355–377. Springer, 2003.