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 $\sim$ 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 $\delta$ (for definitions) and $\iota$ (for pattern-matching on inductive data) rules and hold computationally just like the more common $\beta$-rule.

The second is the judgmental equality ($\equiv$): it is typed, tractable for a machine to decide and typically includes $\eta$-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 ($\equiv$): 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 $\nu$-rules. We gain, for example, that $\text{map } \text{swap} \cdot \text{map } \text{swap} \equiv \text{id}$, where swap swaps the elements of a pair.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\text{map} : (a → b) → list a → list b  \\
\text{map } f \; [] \rightsquigarrow []  \\
\text{map } f \; (x :: xs) \rightsquigarrow f x :: \text{map } f \; xs  \\
(++) : list a → list a → list a  \\
[] ++ ys \sim ys  \\
x :: xs ++ ys \sim x :: (xs ++ ys)  \\
\hline
\end{tabular}
\caption{δι-rules - computational}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\text{Γ} \vdash f = \lambda x. f x : a → b  \\
\text{Γ} \vdash p = (\pi_1 \; p, \; \pi_2 \; p) : a \times b  \\
\text{Γ} \vdash u = () : 1  \\
\hline
\end{tabular}
\caption{η-rules - canonicity}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\text{a} & \text{π}_1 & \text{π}_2 & ++ \; \text{ys} & \text{map } f \; \text{fold } n \; c  \\
\hline
\text{xs} ++ [] & = & \text{xs}  \\
\text{(xs} ++ \; \text{ys}) ++ \; \text{zs} & = & \text{xs} ++ (\text{ys} ++ \; \text{zs})  \\
\text{map } \text{id } \text{xs} & = & \text{xs}  \\
\text{map } f \; \text{map } g \; \text{ks} & = & \text{map } (f \cdot g ) \; \text{ks}  \\
\text{map } f \; (\text{ks} ++ \; \text{ys}) & = & \text{map } f \; \text{ks} ++ \; \text{map } f \; \text{ys}  \\
\text{fold } c \; n \; (\text{map } f \; \text{ks}) & = & \text{fold } (c \cdot f) \; n \; \text{ks}  \\
\text{fold } c \; n \; (\text{ks} ++ \; \text{ys}) & = & \text{fold } c \; (\text{fold } c \; n \; \text{ys}) \; \text{ks}  \\
\hline
\end{tabular}
\caption{ν-rules}
\end{table}
A $\nu$-rule is an equation between neutral terms with the same
nut which holds just by structural induction on the nut, with $\beta_\nu$
reducing subgoals to inductive hypotheses – the classic proof pat-
ttern of Boyer and Moore [14]. Consequently, we need only use
$\nu$-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 success-
ful experiment in formalizing a modified normalization by evalua-
tion proof for simply-typed $\lambda$-calculus with list primitives and the
$\nu$-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 unrel-
ated to the matter at hand: Daniëllsson’s formalization of a Type
Theory as an inductive-recursive family uses a non strictly posi-
tive datatype [21]. Abel et al. [2] resort to recursive domain equa-
tions together with logical relations proving them meaningful,
McBride’s proposition [12] 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 Strath-
clyde [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.

$$\sigma, \tau, \ldots ::= \alpha_i \mid '1' \mid \sigma \times \tau \mid \sigma \to \tau \mid '\text{list } \sigma'$$

Remark In the Agda implementation this indexing by a finite set
of base types is modelled by defining a nat-indexed family type$_n$
with a constructor $'\alpha$ taking a natural number $k$ bounded by $n$ (an
element of 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.

$$t, u, \ldots ::= \lambda x.t \mid t \mathbin{\text{\&}} u \mid '()' \mid t \mathbin{\text{\&}} \ '(' \mid u \mathbin{\text{\&}} '()' \mid u \mathbin{\text{\&}} '()' \mid u \mathbin{\text{\&}} '()' \mid \text{hd } t \mathbin{\text{\&}} '\text{map}(f, x)\mid xs \mathbin{\text{\&}} y \mathbin{\text{\&}} \text{fold}(c, n, xs)$$

For sake of clarity in the formalization, we quote the construc-
tors 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 treat-
ment here, we always assume freshness of the variables introduced
by $\lambda$-abstractions. And we do not artificially separate well-typed
terms and typing derivations; in other words we will use alternati-
vably $\Gamma \vdash t : \sigma$ and $t : \Gamma \vdash \sigma$ to denote the same objects.

Weakening The notion of context inclusion gives rise to a weak-
ening operation $\text{weak}$, which can be viewed as the action on mor-
phisms of the functor $\vdash \sigma$ from the category of contexts and their
inclusions to the category of well-typed terms and functions be-
tween 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 in-
tensional 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 $\Gamma$ and $\Delta$, the well-typed parallel substitution from $\Gamma$ to $\Delta$ is defined by:

$$\Delta \triangleright^\rho \Gamma = \left\{ \begin{array}{ll}
\top & \text{if } \Gamma = \varepsilon \\
\Delta \triangleright^\rho \chi \times \Delta \vdash \sigma & \text{if } \Gamma = \varepsilon \cdot \chi \vdash (x : \sigma)
\end{array} \right.$$

We write $t[\rho]$ for the application of the parallel substitution
$\rho : \Delta \triangleright^\rho \Gamma$ to the term $t : \Gamma \vdash \sigma$ 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: $^\triangleright$) for the extension and the original concept.

Judgmental Equality The equational theory of the calculus, de-
noted $\equiv_{\beta_\nu\delta}$, is quite naturally the congruence closure of the
$\beta_\nu\delta$-rules described earlier where reductions under $\lambda$-abstrac-
tions are allowed. In this paper, we also mention the relation $\sim_{\beta_\nu\delta}$
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 normal-
ization 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 the-
ory. 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 func-
tion building elements of the model from well-typed terms per-
forms $\beta\delta$-reductions and does not reduce under $\lambda$-abstractions
effectively building closures – using the $\lambda$-abstractions of the host
language – when encountering one. Secondly the quoting machin-
ery extracting terms from the model performs $\eta$-expansions where
needed which will cause the closures to be reduced and new com-
putations 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 $\beta$-normal form may be transformed into
long $\beta$-normal form by suitable $\eta$-expansions. Therefore
each term $r$ may be transformed into a unique long $\beta$-
normal form $r^\eta$ by $\beta$-conversion and $\eta$-expansions.

Building on this asertainment, we construct a three (rather than
two) staged process successively performing $\beta_\nu\delta$, $\eta$ and finally $\nu$
reductions whilst always potentially calling back a procedure from
a preceding stage to reduce further non-normal terms appearing
when e.g. going under $\lambda$-abstractions during $\eta$-expansion, distribut-
ing 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
We will consider the normalization of Example nil or cons, then it is reduced to it by the first step of the reduction.

In our case) is convertible to a constructor headed term (be it either terms which are themselves stuck. In other words: if a term (a list

Remark It should be noted that the two last steps never reduce a

Example We will consider the normalization of (λx.x).nil of type ε ⊢ list (1 × αk)→list (1 × αk) as a running example demonstrating the successive steps.

Untyped βη-reductions The first intermediate language we are going to encounter is composed of weak-head βη-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.

Figure 1: Context inclusion and typing rules

\[
\begin{align*}
\text{base: } & \varepsilon \subseteq \varepsilon \\
\text{pr: } & \Gamma \subseteq \Delta \\
\text{pop: } & \Gamma \subseteq \Delta \cdot (x: \sigma) \\
\text{step: } & \Gamma \subseteq \Delta \cdot (x: \sigma)
\end{align*}
\]

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 λ; and calls an helper function (e.g. wh-S, wh-π₁, wh-π₂, 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 (λx.x).nil to the usual identity function: λ 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.

\[
\begin{align*}
\text{n := } & x | n \cdot v | \pi_1 n | \pi_2 n | \text{fold}(v_1, v_2, l) \\
v := & n_{\lambda k} | l | \lambda x.v | \varepsilon | v_1 | v_2 | [] | v_1 ::= v_2 \\
l := & n_{\text{list } \sigma} | \text{map}(f, v) | l | ++ | v
\end{align*}
\]

Figure 3: η-long values

The η-expansion of product and function type actually calls back the subroutines for βη-rule 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 ηnorm the main function performing this step and present it in Figure 4 ηlist and ηneut are two trivial auxiliary functions going structurally through either lists or neutral terms and calling ηnorm whenever necessary.

\[
\begin{align*}
\eta\text{norm}(\alpha_k) & = t = \eta\text{neut} t \\
\eta\text{norm}(\text{list } \sigma) & = \eta\text{list } \sigma t \\
\eta\text{norm}(1) & = \varepsilon \\
\eta\text{norm}(\sigma \rightarrow \tau) & = \eta\text{norm } \sigma (\text{wh-π₁ } t), \eta\text{norm } \tau (\text{wh-π₂ } t) \\
\eta\text{norm}(\sigma \rightarrow \tau) & = \lambda x. \eta\text{norm } \tau (\text{wh-Σ } x)
\end{align*}
\]

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

Example The η-expansion of the evaluated form λ tt | x.x of type ε ⊢ list (1 × αk)→list (1 × αk) proceeds in multiple steps.

- The arrow type forces us to introduce a λ-abstraction: λ x. ηnorm (list (1 × αk)) ((λ tt | x.x) wh-Σ x).
- Now, (λ tt | x.x) wh-Σ x trivially reduces to x, a neutral of list type, left unmodified by η-expansion. Hence the η-long form: λ tt | x.x.
\textbf{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 \(\text{map}(.).+\) – referred to as “mapp” – has the obvious semantics that it represents the concatenation of a stuck map and a list.

\begin{align*}
  n &::= x | n \cdot s | \tau_1 n | \tau_2 n | \text{fold}(v_1, v_2, n) \\
  v &::= n \cdot s | s \cdot \lambda x. v | \text{map}(.).+ | v_1 \cdot | v_1 \cdot : v_2 \\
  s &::= \text{map}(v_1, n).++ v_2
\end{align*}

\textbf{Figure 5: Standard Forms}

The standard lists \(s\) are produced by flattening the stuck map \(l\) append trees present in \(l\) after the end of the previous procedure whilst the fold \(l\) map and fold \(l\) 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 \(\text{nf-norm}, \text{nf-neut}\) and \(\text{nf-list}\) respectively turning \(η\)-long normals, neutrals and lists into elements of the corresponding standard classes. \(\text{nf-norm}\) and \(\text{nf-neut}\) are mostly structural except for the few cases described in Figure 6.

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

\begin{align*}
  \text{nf-norm}(\text{list } s) &\sigma x_{ne} = \text{nf-list } xs \\
  \text{nf-neut}(\text{fold } cn x s) &\sigma = \text{nf-fold } cn (\text{nf-list } xs) \\
  \text{nf-list } x_{ne} &\sigma = \text{map}(\text{nf-norm}(\lambda x. x), x s) .++. \text{[]} \\
  \text{nf-list}(\text{map } f x s) &\sigma = \text{nf-map } f (\text{nf-list } x s) \\
  \text{nf-list}(x s .++. ys) &\sigma = \text{nf-}(\text{nf-list } x s)(\text{nf-norm } \lambda y s) \\
  \text{nf-fold } cn (\text{map } f x s) .++. ys &\sigma = \text{fold } cf \text{ ih } xs \\
  \text{where } &cf = \text{standard } (\text{wh-} \text{cf} ) \\
  \text{ih} &\sigma = \text{standard } (\text{wh-fold } cn ys) \\
  \text{nf-map } f (\text{map } g x s) .++. ys &\sigma = \text{map}(f g, x s) .++. \text{fys} \\
  \text{where } &fg = \text{standard } (\text{fwh-} \text{fg} ) \\
  \text{fys} &\sigma = \text{standard } (\text{wh-map } f y s) \\
  \text{nf-}(.+)(\text{map } f x s) .++. ys &\sigma = \text{map}(f x s) .++. \text{yzs} \\
  \text{where } &yzs = \text{standard } (\text{yswh-} \text{ysz} )
\end{align*}

\textbf{Figure 6: From \(η\)-long values to standard ones}

\textbf{Example} \(\text{nf-norm}\) does not touch the \(λ\)-abstraction but expands the neutral \(x\) of type \(\text{list}(\lambda x. a)\) to \(\text{map } \text{id}, x\) .++. \[\text{[]}\] where \(\text{id}\) is the normal form of the identity function on \(\lambda x. a\).

We leave it to the reader to check that:

\begin{align*}
  \text{id} &\sigma = \text{η-norm } (\lambda x. a_k) p \\
  \text{id} &\lambda p. \text{η-norm } (\lambda x. a) p \\
  &\lambda p. (\text{η-norm } \lambda (\tau_1 p), \text{η-norm } a_k (\tau_2 p)) \\
  &\lambda p. (\text{[][]} , \text{[][]} )
\end{align*}

Hence the final standard form of \(\text{map}(\lambda x. x)\):

\begin{align*}
  \lambda x. \text{map}(\lambda p. (\text{[][]} , \text{[][]} ) , x s) .++. \text{[]} \\
\end{align*}

\textbf{Remark} We will call \(Γ \vdash_{nf} σ\) the typing derivations restricted to standard values as per the previous section’s definitions and \(Γ \vdash_{ne} σ\) 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.

\textbf{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.

\footnote{E.g. \(n + 0 \sim n\) in a calculus where + is defined by case analysis on the first argument and this expression is therefore stuck.}

\section{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.

\textbf{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:

Dannys’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 [11] 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] built extensions able to deal with a variety of type theories. Last but not least Ahman and Staton [3, 4] explained how to treat calculi equipped with algebraic effects which can be seen as an extension of the calculus of Watkins et al. [19] extending judgmental equality with equations for concurrency or Filinski’s computational λ-calculus [25].

\begin{thebibliography}{99}


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 \( \eta \)-rules for unit, product and arrow types; therefore the semantical counterpart of such types need not be more complex than unit, pairs and actual function spaces.

\[
\begin{align*}
\mathcal{M}(\Gamma, -) & : \text{type}_n \rightarrow \text{Set} \\
\mathcal{M}(\Gamma, 1) & = I
\end{align*}
\]

\[
\begin{align*}
\mathcal{M}(\Gamma, \alpha_k) & = \Gamma \vdash \alpha_k \\
\mathcal{M}(\Gamma, \sigma \times \tau) & = \mathcal{M}(\Gamma, \sigma) \times \mathcal{M}(\Gamma, \tau) \\
\mathcal{M}(\Gamma, \text{\ list } \tau) & = \forall \Delta, \Gamma \subseteq \Delta \rightarrow \mathcal{M}(\Delta, \sigma) \rightarrow \mathcal{M}(\Delta, \tau) \\
\mathcal{M}(\Gamma, \text{\ list } \tau) & = \mathcal{L}(\Gamma, \sigma, \mathcal{M}(\cdot, \cdot))
\end{align*}
\]

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 fold-fusions.

\[
\begin{align*}
\Gamma : \text{Con}(\text{type}_n) & \quad \sigma : \text{type}_n \quad M_\sigma : \text{Con}(\text{type}_n) \rightarrow \text{Set} \\
\sigma & = \text{Con}(\text{type}_n) \rightarrow \text{Set} \\
\sigma & = \text{Con}(\text{type}_n)
\end{align*}
\]

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

Unit, base and product types The unit case is trivial: the reification process returns \( \text{\ Th} \) while the reflection one produces the only inhabitant of \( \top \). The base 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 \( \eta \)-expansion of the stuck term. We can now focus on the more subtle cases.

Arrow type The function case is obtained by \( \eta \)-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.

\[
\begin{align*}
\downarrow_{nf} & \lambda x. \tau F \defeq \lambda x. \uparrow_{ne} F \downarrow_{nf} \tau x \\
\uparrow_{ne} & \lambda x. \tau F \defeq \Delta \text{ inc } x. \uparrow_{ne} (\text{wkns}(f) \ \text{ inc } x) \ \downarrow_{nf} \tau x
\end{align*}
\]

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 \( \downarrow_{nf} \) and \( \uparrow_{ne} \) for the helper functions performing reification and reflection on lists of type \( \text{\ list } \sigma \).

\[
\begin{align*}
\downarrow_{nf} [\ ] & \defeq \Delta \\
\uparrow_{ne} HD \downarrow_{nf} :: TL & \defeq \Delta \\
\downarrow_{nf} \text{map}(f, [x]) & \defeq \text{map}(\lambda x. \downarrow_{nf} f(x), [x]) \\
\uparrow_{ne} HD \downarrow_{nf} :: TL & \defeq \uparrow_{nf} TL \\
\downarrow_{nf} \text{map}(f, [x]) & \defeq \text{map}(\lambda x. \uparrow_{ne} f(x), [x]) \\
\uparrow_{ne} HD \downarrow_{nf} :: TL & \defeq \uparrow_{nf} TL \\
\end{align*}
\]

This injection corresponds to applying the identity function and monoid law. Indeed \( \Delta, \Gamma \subseteq \Delta \rightarrow \Delta \vdash_{ne} \sigma \rightarrow \mathcal{M}(\Delta, \sigma) \) to fit in the semantical list mapp constructor.

\[
\begin{align*}
\text{map}(\lambda \Delta. \uparrow_{ne} \sigma, [x]) & \defeq [\text{map}(\lambda \Delta. \uparrow_{ne} \sigma, x)]
\end{align*}
\]

Example of \( \eta \)-expansions provoked by the reflect / reify functions; for \( x \) a neutral list of type \( \text{\ list } (1 \times \text{\ alpha}) \), we get an expanded version by drowning it in the model and reifying it back:

\[
[1 \times \text{\ list } (1 \times \text{\ alpha})] \rightarrow ([1 \times \text{\ list } (1 \times \text{\ alpha})]) \rightarrow [\text{map}(\lambda p. (\text{\ Th } \text{\ Th}), p), [x]]
\]

This showcases the \( \eta \)-expansion of unit, products and functions as well as the use of the identity laws mentioned during the definition of \( \uparrow_{ne} \).

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 \( \mathcal{M}'(\Delta, \Gamma) \). Indeed the definition of the reflection function \( \uparrow_{ne} \) together with the existence of environment weakenings give us the necessary machinery to produce a diagonal semantical environment \( \mathcal{M}'(\Gamma, \Gamma) \) 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 \( \mathcal{M}'(\Delta, \Gamma) \), one can build a semantical object in \( \mathcal{M}(\Delta, \sigma) \).

Proof. A simple induction on the term to be evaluated using the semantic 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 \( \Gamma \vdash \tau \), an evaluation environment \( R \) in \( \mathcal{M}'(\Delta, \Gamma) \) and we are given a context \( E \), a proof \( \text{inc} \) that \( \Delta \subseteq E \) and an object \( S \) living in \( \mathcal{M}(E, \sigma) \). By combining \( S \) and a weakening of \( R \) along \( \text{inc} \), we get an evaluation environment of type \( \mathcal{M}'(E, \Gamma, \sigma) \) 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, reification 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
be observed in other variants of normalization by deciding more exotic equational theories e.g. having \( \eta \)-reduction but no \( \eta \)-rules for the simply-typed \( \lambda \)-calculus [14].

**Remark** The only place where type information is needed is when reorganizing neutrals following \( \nu \)-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 1.1.1 we describe all the computations eagerly for Agda to see the termination argument but a subtle evaluation strategy applied to the produced code could reform the behaviour of the layered approach. It would have to form lambda closures in the arrow case, fire eagerly only the reductions eliminating of the layered approach. It would have to form lambda closures ready well explained by Catarina Coquand [18] in her presentation completeness theorem tightening the specification of the procedure. Hence leading to a natural interpretation:

\[
\frac{A B C}{F(t) \rightsquigarrow A \times B \times C}
\]

### 5.1 Soundness

Soundness amounts to re-building the propositional part of the reducibility candidate argument [25] which has been erased to get the bare bones model. The logical relation \( M(\Gamma, \sigma) \ni t \vdash T \) relates a semantical object \( T \) in \( M(\Gamma, \sigma) \) and a term \( t \) in \( \Gamma \vdash \sigma \) which is morally the source of the semantical object.

**Logical Relation for Soundness** \( M(\Gamma, \sigma) \ni t \vdash T \) is defined by induction on the type \( \sigma \) plus an appropriate inductive definition for the list case \( \mathcal{L}(\Gamma, \sigma, M_a, M_\tau, M_s, \vdash, \vdash) \ni t \vdash \mathcal{S} \). Here are the formation rules of these types.

\[
t: \Gamma \vdash \sigma \quad T: M(\Gamma, \sigma) \quad M(\Gamma, \sigma) \ni t \vdash T: \mathcal{S}
\]

\[
xs: \mathcal{L}(\Gamma, \sigma, M_a, M_\tau, M_s, \vdash, \vdash) \ni t \vdash \mathcal{S} \quad \mathcal{S}: \mathcal{S}
\]

**Remark** It should be no surprise to the now experienced reader that the inductive definition of the logical relation for \( \mathcal{L} \) is defined by 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:

\[
\frac{M(\Gamma, \sigma) \ni t \vdash T \quad M(\Gamma, \sigma) \ni t \vdash T}{M(\Gamma, \sigma) \ni t \vdash \mathcal{T}}
\]

\[
\frac{a: \Gamma \vdash \sigma \quad b: \Gamma \vdash \tau}{M(\Gamma, \sigma) \ni a \vdash A \quad M(\Gamma, \sigma) \ni b \vdash B}
\]

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 type is a correct normal form for any input of type list but it certainly is not a satisfactory answer with respect to \( \beta \eta \mu \nu \)-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 \( M \) 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

\[
\frac{A \quad B \quad C}{F(t) \rightsquigarrow A \times B \times C}
\]
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:

\[
\frac{t \rightsquigarrow^{\ast}_{\Delta}}{\text{L}(\Gamma, \sigma, \Delta, \tau) \ni t \Downarrow^{\ast} F}
\]

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 semantics function to behave like the one discovered.

\[
\frac{t \rightsquigarrow^{\ast}_{\Delta}}{\text{L}(\Gamma, \sigma, \Delta, \tau) \ni t \Downarrow^{\ast} F}
\]

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 i.e. the semantical object indeed is a reduct of the source term.

Pointwise extension

We denote by \(\mathcal{M}^T(\cdot, \cdot) \ni \cdot \Downarrow \cdot \) 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 \(t =_{\text{ne}} \sigma\) then \(\mathcal{M}(\Gamma, \sigma) \ni t =_{\text{ne}} \sigma\).
2. If \(t =_{\text{ne}} \sigma\) then \(\mathcal{M}(\Gamma, \sigma) \ni t =_{\text{ne}} \sigma\).

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 \(s =_{\text{ne}} t\) in \(\Gamma =_{\text{ne}} \sigma\), if \(s \rightsquigarrow^{\ast}_{\Delta} \) then for all \(T =_{\text{ne}} \sigma\) such that \(\mathcal{M}(\Gamma, \sigma) \ni t =_{\text{ne}} \sigma\), it is also true that \(\mathcal{M}(\Gamma, \sigma) \ni s =_{\text{ne}} \sigma\).

The proof of soundness then mainly involves showing that the semantic counterparts of the language’s combinator’s we defined during the model construction are compatible with the logical relation. Namely that e.g. if \(\mathcal{M}(\Gamma, \sigma) \ni t =_{\text{ne}} \sigma\) and \(\mathcal{M}(\Gamma, \sigma) \ni t =_{\text{ne}} \sigma\), then it is also true that \(\mathcal{M}(\Gamma, \sigma) \ni s =_{\text{ne}} \sigma\).

**Theorem 5.3.** Given a term \(t\) : \(\Gamma =_{\text{ne}} \sigma\), a parallel substitution \(\rho : \Delta \ni \Gamma\) and an evaluation environment \(R\) such that \(\rho \text{ and } R\) are related \((\mathcal{M}(\Delta, \Delta) \ni \rho \Downarrow R\) holds), the evaluation of \(t\) in \(R\) is related to \(t[r] : \mathcal{M}(\Delta, \sigma) \ni t[r] \Downarrow R\).

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 \(\mathcal{M}^T(\Delta, \Gamma) \ni \rho \Downarrow \rho\).

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 \(t = \lambda x. b\), we are given a context \(E\) together with a proof \(\text{inc}\) that it is an extension of \(\Delta\), a term \(u\) and an object \(U\) which are related \(\mathcal{M}(E, \sigma) \ni u \Downarrow U\) and, finally, a term \(s : E =_{\text{ne}} \tau\) which reduces to \((\lambda x. b)[\rho] \Downarrow u\). First of all, we should notice that \(s \rightsquigarrow^{\ast}_{\Delta} \) and therefore that to prove \(\mathcal{M}(E, \tau) \ni s \Downarrow T\) it is enough to prove that \(\mathcal{M}(E, \tau) \ni b[\rho, x \mapsto u] \Downarrow T\). And we get just that by using the induction hypothesis with the related parallel substitution \(\rho\) and evaluation environment \(R\) obtained by the combination of the weakening of \(\rho\) (resp. \(U\)) along \(\text{inc}\) with \(u\) (resp. \(U\)).

**Corollary 5.4.** A term \(t\) reduces to the normal form produced by the normalization by evaluation procedure: \(t \rightsquigarrow^{\ast}_{\Delta} \text{norm} t\).

Proof. The identity parallel substitution is related to the diagonal evaluation environment and \(t[1\Delta]\) is equal to \(t\) hence, by the previous theorem, \(\mathcal{M}(\Gamma, \sigma) \ni t \Downarrow \text{eval}(t[1\Delta], t)\) and then \(t \rightsquigarrow^{\ast}_{\Delta, \sigma} \text{norm} t\).

5.2 Completeness

Completeness can be summed up by the fact that the interpretation of \(\Delta\)-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 \(T =_{\text{ne}} U\) while \(T \equiv_{\text{ne}} U\) being uniform is written \(\text{Un}_{\text{ne}} T\).

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 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 \(\text{list } \sigma\) case, extensionality is an inductive set basically building the (semantical) diagonal relation on lists of the same type. It is parametrized by a relation \(E\) on terms of type \(\mathcal{M}(\Delta, \sigma)\) (for any context \(\Delta\)) which is, in the practical case instantiated with \(\Delta =_{\text{ne}} \Delta\) as would one 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 \(\text{Un}_{\text{ne}}\), 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.
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 \( \alpha; T \equiv_\sigma \alpha; U \)
2. If \( \text{inc} \) is a neutral \( \Gamma \vdash_{\sigma} \) \( \sigma \) then \( \text{Uni}_\sigma (\alpha; \text{inc}) \)
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 \( s \equiv_{\delta_{\beta \eta \nu}} t \) and if \( R \) is a uniform environment in \( \mathcal{M}^0(\Delta, \Gamma) \) then \( \text{eval}(s, R) \equiv_\sigma \text{eval}(t, R) \).

**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.

**\( \beta \)-rules** Each one the \( \iota \) rules holds by reflexivity of the extensional equality, indeed evaluation realizes these computation rules syntactically. The case of the \( \beta \) 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 evaluation 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 \( \text{eval} \) 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 \( \iota \) 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 \( s \) and \( t \) of type \( \Gamma \vdash_{\sigma} \), if \( s \equiv_{\delta_{\beta \eta \nu}} \) \( t \) then \( \text{norm} t = \text{norm} u \).

**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 \( t \equiv \beta\eta nv u \), 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 \( \eta \)-rules is decidable.

Proof. Given terms \( t \) and \( u \) of the same type \( \Gamma \vdash \sigma \), we can get two normal forms \( t_{nf} = \text{norm} t \) and \( u_{nf} = \text{norm} u \) and test them for equality up-to \( \alpha \)-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 \( \eta \)-rules

1. In a context with two functions \( f \) and \( g \) of type \( \sigma \xrightarrow{\cdot} 1 \), \( \lambda x. x \cdot \text{map}(f, x) \) and \( \lambda x. x \cdot \text{map}(g, x) \) both normalize to \( \lambda x. \text{map}(\lambda .(-), x) \xrightarrow{\cdot} 1 \) and are therefore declared equal.

2. At type \( \Gamma \vdash \text{list} \left( \text{\hat{\alpha} k \times \hat{\alpha} j} \right) \rightarrow \text{list} \left( \text{\hat{\alpha} k \times \hat{\alpha} j} \right) \), the terms \( \lambda x. x \cdot \text{map}(\text{swap}, \text{map}(\text{swap}, x)) \) and \( \lambda x. \text{map}(\lambda p. (p \cdot \text{swap} \cdot \text{\hat{\alpha} k}), x) \) are not convertible. Indeed, if they were then the completeness result guarantees us that \( \lambda x. x \cdot \text{map}(\text{swap}, \text{map}(\text{swap}, x)) \) and \( \lambda x. \text{map}(\lambda p. (p \cdot \text{swap} \cdot \text{\hat{\alpha} k}), x) \) would be equal which leads to a contradiction.

6. Scaling up to Type Theory

Now that we know for sure that the judgmental equality can be safely extended with some \( \eta \)-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 \( \Pi \)-types and product types by \( \Sigma \)-types but the basic machinery of evaluation and type-directed \( \eta \)-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. \( \Gamma \vdash t \equiv C \cdot t' \) should imply that \( t \xrightarrow{\cdot} C \cdot t' \). This can be enforced by making sure that candidates for \( \nu \)-rules are only reorganizing spines of stuck eliminators and are absolutely never emitting new constructors.

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

\( \begin{align*}
\Gamma \vdash f : F \\
\Gamma \vdash s : S' \\
\Gamma \vdash f : T[s/x]
\end{align*} \)

\( \begin{align*}
F \xrightarrow{\cdot} (x : S) \rightarrow T \\
\vdash F \rightarrow (x : S) \rightarrow T \\
\vdash S \rightarrow T \\
\vdash s : S \\
\vdash s : S
\end{align*} \)

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 \( n \cdot \text{refl} \cdot n : N = N \) 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 \( \nu \)-rule which discards coercions whenever it is type-safe to do so:

\[
\mathbb{F} [Q : S = T] = \mathbb{U} \quad \text{if } S \equiv T : \text{Set}
\]

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 \( \nu \)-rules in this paper. There, as here, this spares the evaluation process from decisions which involve \( \eta \)-expansion and thus require a name supply. The \( \nu \)-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 parameterized inductive datatypes by modifying the \( \iota \)-rules of their underlying type theory \[14\]. We should very much hope to achieve the same result, as we did here in the special case of lists, just by adding \( \nu \)-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 Dangad 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 \( \nu \)-rules:

\[
\mathbb{F} [\text{return}] = \mathbb{U} \quad \text{if } \text{return} \equiv \sigma \Rightarrow \rho \quad \text{if } \rho \equiv (\tau \Rightarrow \sigma) \cdot \rho
\]

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

**Decomposing functors.** Dangad 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

\[
\text{Decode} : \text{IDesc} I \rightarrow (I \rightarrow \text{Set}) \rightarrow \text{Set}
\]

is structurally recursive in the description and lifts pointwise to an interpretation of substitutions in the IDesc monad

\[
\llbracket \_ \rrbracket : (O \rightarrow \text{IDesc} I) \rightarrow (I \rightarrow \text{Set}) \rightarrow (O \rightarrow \text{Set})
\]

\[
\llbracket \sigma \rrbracket (X) o = \text{Decode} \ (\sigma \ o) X
\]

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 \( \beta \delta \alpha \), but we may also interpret Kleisli composition as reverse functor composition

\[
(\llbracket \_ \rrbracket \Rightarrow \sigma) \cdot \rho] = [\rho] \cdot [\sigma]
\]

by means of a \( \nu \)-rule

\[
\text{decode} \ [\text{id}] \Rightarrow \sigma X = \text{decode} \ [\text{id}] \ ([\sigma] X)
\]

taking each \( D \) to be some \( \rho \circ o \). 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 \( (\llbracket \_ \rrbracket \Rightarrow \sigma) \cdot \rho \), where \( \rho \) describes the invariant superstructures and \( \sigma \) the modified substructures, then invoke the functoriality of \([\rho]\). This \( \nu \)-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 Lammel 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

\[
\nu \ell_1 : \text{Set} \\
\ell_1 : \text{Set} \\
\nu \ell_1 : \text{Set} \\
\ell_2 : \text{Set} \\
\nu \ell_2 : \text{Set} \\
\ell_2 : \text{Set}
\]

Palmgren \[35\] suggests that one way to model a cumulative hierarchy of such universes is to give each a code in the next, so

\[
[\nu \ell_1 : \text{Set}] \\
[\ell_1 : \text{Set}] \\
[\nu \ell_1 : \text{Set}] \\
[\ell_2 : \text{Set}] \\
[\nu \ell_2 : \text{Set}] \\
[\ell_2 : \text{Set}]
\]

and then define an embedding recursively

\[
\downarrow \ell_1 : \ell_2 \rightarrow \ell_1 \quad \uparrow \ (\ell_1, \ell_2) = \nu \ell_2 \ (\ell_1, \ell_2) (\lambda s. \ \uparrow \ (T s))
\]

but a small frustration with this proposal is that \( s \) is abstracted at type \( \ell_2 (\ell_1, \ell_2) \), but used at type \( \ell_1, \ell_2 \), and these two types are not definitionally equal for an abstract \( S \). One workaround is to make \( \uparrow \) a constructor of \( \ell_2 \), at the cost of some redundancy of representation, but now we might also consider fixing the discrepancy with a \( \nu \)-rule

\[
\ell_2 (\ell_2, \ell_2) \rightarrow \ell_1
\]

This is peculiar for our examples thus far, in that the \( \nu \)-rule is needed even to typecheck the \( \delta \eta \)-rules for \( \downarrow \), reflecting the fact that \( \downarrow \) should not be any old function from \( \ell_1, \ell_2 \) to \( \ell_2 \), but rather one which preserves the meanings given by \( \ell_1, \ell_2 \). In effect, the \( \nu \)-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 \( \nu \)-rules hold of the operations we implement.

**Non-examples.** A key characteristic of a \( \nu \)-rule is that it is a neutral-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 \( x + \text{Suc} \ y = \text{Suc} \ (x + y) \) and \( x \cdot 0 = 0 \) 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 \[18\]. Her untyped rewriting approach sits awkwardly with \( \eta \)-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 \( \nu \)-rules can safely be added is of the utmost importance. We want to let the
programmer negotiate the new $\nu$-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 $\nu$-rules with more prac-
tical presentations of normalization. For instance Grégoire and
Leroy’s conversion by compilation to a bytecode machine derived
from Ocaml’s ZAM \cite{27} decides $\eta$ by expansion only when pro-
voked by a $\lambda$: such laziness is desirable when possible but causes
trouble with $\eta$-rules for unit types and may conceal the potential to
apply $\nu$-rules. Hereditary substitution \cite{19} formalized by Abel \cite{11} and by Keller and Altenkirch \cite{29}, may be easier to adapt.

Acknowledgements

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

References

\begin{enumerate}
\item D. Ahman and S. Sam. Normalization by evaluation and algebraic ef-
fects. In 29th Conference on Mathematical Foundations of Program-
\item P. Boutillier. Equality for lambda-terms with list primitives. *Internship report*, University of Strathclyde, ENS Lyon, July 2009.
\item INRIA. The Coq proof assistant.
\end{enumerate}