Forge crowbars, Acquire normal forms.

G. Allais
guillaume.allais@strath.ac.uk

December 19, 2012

Abstract

Normalization by evaluation is a way to exploit the implementation's language computing capabilities to perform normalization of open lambda terms in a big step fashion. Using different languages or different *crowbars* to leverage their respective reduction mechanisms, one can therefore get different degrees of normalization; be it either the reduction strategies considered or the shape of what ought to be a normal form that varies.

In this document, we build different models (these are the crowbars we make use of to attain the available computing resources) in which we can give denotational semantics of our programs. We then characterize the equational theories they decide.

The implementation of the described material is available on github:
`https://github.com/gallais/agda-nbe`

This work is to be read with a constructive setting in mind: definitions correspond either to datatype declarations or total functions in an appropriate intensional type-theory (e.g. Martin Löf's type theory[17]) while proofs are algorithm turning their hypothesis into their conclusions. As a consequence, somme lemmas will be named and used as functions later on.

In particular, we have formalized all the notions and proofs described here in Agda[19], a proof assistant based on Martin Löf's type theory.

1.1 The calculus

The studied calculus is pretty similar to Pierre Boutillier's[7]; the main difference being the addition of a fold operation as well as append for lists and the extension of the equational theory to support map-fold-append interactions.

Definition *Contexts* are snoc lists with constructors ε for the empty context and $\_ \cdot \_$ for extending and already existing context. The notion of order-sensitive *context inclusion* is defined inductively by the following rules.

$$\frac{}{\mathtt{base} : \varepsilon \subseteq \varepsilon} \qquad \frac{pr : \Gamma \subseteq \Delta}{\mathtt{pop!}\, pr : \Gamma \cdot \sigma \subseteq \Delta \cdot \sigma} \qquad \frac{pr : \Gamma \subseteq \Delta}{\mathtt{step}\, pr : \Gamma \subseteq \Delta \cdot \sigma}$$

Inclusion obviously is reflexive and transitive and these notions play nicely together: contexts and proofs of inclusion form a category.

Definition This simply-typed calculus is equipped with finite types indexed by (a finite set of) base types they are allowed to mention, containing unit and closed under product, list and arrow. This is exactly what the following formation rules describe:

$$\frac{n : \mathbb{N}}{\mathtt{type}_n : \mathtt{Set}} \qquad \frac{k : \mathtt{Fin}_n}{\text{`}\mathtt{b}\ k : \mathtt{type}_n} \qquad \frac{\sigma : \mathtt{type}_n \qquad \tau : \mathtt{type}_n}{\sigma\ \text{`}\times\ \tau : \mathtt{type}_n}$$

$$\frac{}{\text{`}\mathtt{1} : \mathtt{type}_n} \qquad \frac{\sigma : \mathtt{type}_n}{\text{`}\mathtt{list}\ \sigma : \mathtt{type}_n} \qquad \frac{\sigma : \mathtt{type}_n \qquad \tau : \mathtt{type}_n}{\sigma\ \text{`}\rightarrow\ \tau : \mathtt{type}_n}$$

Definition *Terms* are described as correct typing derivations in an intuitionistic sequent calculus fashion.

$$\frac{\Gamma : \mathtt{Con}(\mathtt{type}_n) \qquad \sigma : \mathtt{type}_n}{\Gamma \vdash \sigma : \mathtt{Set}}$$

On top of the usual (potentially infix) constructors for variables (`v), lambda abstractions (`λ), application (`$), lists (`[] and `:: ), pairs ( `, and the corresponding projections `$\pi_1$ and `π_2), the calculus has special constructors for distinguished operations:

$$\frac{xs\ ys \colon \Gamma \vdash \text{`}\mathtt{list}\ \sigma}{xs\ \text{`}{+\!+}\ ys \colon \Gamma \vdash \text{`}\mathtt{list}\ \sigma} \qquad \frac{f \colon \Gamma \vdash \sigma \text{`}{\to} \tau \qquad xs \colon \Gamma \vdash \text{`}\mathtt{list}\ \sigma}{\text{`}\mathtt{map}(f, xs) \colon \Gamma \vdash \text{`}\mathtt{list}\ \tau}$$

$$\frac{c \colon \Gamma \vdash \sigma \text{`}{\to} \tau \text{`}{\to} \tau \qquad n \colon \Gamma \vdash \tau \qquad xs \colon \Gamma \vdash \text{`}\mathtt{list}\ \sigma}{\text{`}\mathtt{fold}(c, n, xs) \colon \Gamma \vdash \tau}$$

Context inclusions quite naturally induce a functorial operation on terms: weakening. The weakening induced by the identity inclusion is the identity function and composition of weakenings is quite obviously the weakening induced by the composition of the corresponding context inclusions. Throughout the document, they will be mostly left implicit.

The pointwise extension of the well-typed terms' definition to contexts corresponds to the notion of parallel substitution.

$$\Delta \vdash_{\varepsilon} \Gamma = \begin{cases} \top & \text{if } \Gamma = \varepsilon \\ \Delta \vdash_{\varepsilon} \Gamma' \times \Delta \vdash \sigma & \text{if } \Gamma = \Gamma' \cdot \sigma \end{cases}$$

These substitutions are the morphisms of a category whose objects are terms and where composition $\rho \bowtie \rho'$ is the pointwise application of ρ on the terms defining ρ'.

1.2 The notion of equality

The notion of equality used for this calculus is the congruence closure of a typed reduction relation combining computational and standardization steps.

β and ι reductions are concerned with the *computations* triggered by putting a constructor face to face with an eliminator. They are nowhere revolutionary enough to be worth describing here in more details.

η reductions on the other hand deal with *standardizations*; in other words η rules are here to ensure that terms can be given a canonical form mostly determined by their type. They can morally be separated in two sets.

The first set of such rules describes canonical constructors: the term for a function should start with a λ-abstraction hence the typed reduction rule $\Gamma \vdash \sigma \text{`}{\to} \tau \ni t \rightsquigarrow_{\eta} \text{`}\lambda(t\ \text{`}\$\ \text{`}\mathtt{v}\,0)$, the inhabitant of a product type should be a pair therefore $\Gamma \vdash \sigma \text{`}{\times} \tau \ni t \rightsquigarrow_{\eta} (\text{`}\pi_1\, t\ \text{`},\ \text{`}\pi_2\, t)$ just like any inhabitant of unit should be its canonical inhabitant: $\Gamma \vdash \text{`}\mathtt{1} \ni t \rightsquigarrow_{\eta} \mathtt{tt}$.

The second set of rules is a bit less standard: it internalizes equations which are usually left in the propositional equality. They can indeed be proved by a simple induction. The reader might recognizes optimizing transformations (or their converse) used when compiling functional programs. The first batch deals with the functoriality of `'list` and the corresponding lifting operation `'map` acting on functions.

$$\Gamma \vdash \text{`}\mathtt{list}\ \sigma \ni \text{`}\mathtt{map}(g, \text{`}\mathtt{map}(f, xs)) \rightsquigarrow_{\eta} \text{`}\mathtt{map}(g \cdot f, xs)$$

$$\Gamma \vdash \text{`}\mathtt{list}\ \sigma \ni xs \rightsquigarrow_{\eta} \text{`}\mathtt{map}(\mathtt{id}, xs)$$

The second one is taking care of the monoidal structure built by the append function; it internalizes the associativity of `'++` and the existence of a right unit `'[]` (which is also a left unit thanks to ι rules).

$$\Gamma \vdash \text{`list}\ \sigma\ \ni (xs\ \text{`++}\ ys)\ \text{`++}\ zs \rightsquigarrow_\eta xs\ \text{`++}\ (ys\ \text{`++}\ zs)$$
$$\Gamma \vdash \text{`list}\ \sigma\ \ni xs \rightsquigarrow_\eta xs\ \text{`++}\ \text{`[]}$$

Finally the last one explains how the functor interacts with the monoidal structure.

$$\Gamma \vdash \text{`list}\ \sigma\ \ni \text{`map}(f, xs\ \text{`++}\ ys) \rightsquigarrow_\eta \text{`map}(f, xs)\ \text{`++}\ \text{`map}(f, ys)$$

Sanity check

One easy sanity check one can perform is to write an 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. This is done in three steps: one starts by giving an interpretation of types of the calculus as sets in the target type theory; a well-typed interpretation of terms comes next and finally one has to establish a soundness proof showing that elements related by the reduction relation are provably equal in the type theory.

In the present case, it is quite easy to implement this sanity check in an extensional type theory[1] (cf. page 24). Once the reader is confident that the described theory somehow makes sense, she can move on to see how to decide it.

1.3 Deciding equality of terms

The reduction relation given for this calculus clearly is not strongly normalizing because of the presence of various η rules allowing to create new redexes by artifically expanding stuck eliminations[2]. As a consequence, testing equality by iterating reduction steps until normal forms –which can be compared syntactically– are reached is not a solution. There are a couple of different ways to deal with this problem.

One may want to work on pairs of terms and perform successive weak-head reductions on both elements until either the two head symbols are distincts or the equation is trivially true. This technique has been showcased both by Grégoire and Leroy[14] throught compilation of terms to a bytecode machine derived from Ocaml's ZAM and Abel, Coquand and Dybjer[4] whose algorithm works directly on typable terms but is yet to be proven terminating.

Or one can notice that even if the calculus is not strongly normalizing, it does enjoy unique $\beta\iota$-normal η-long forms which can be reached by cleverly applying a deterministic reduction strategy to the terms. It is therefore possible to stick to the approach involving separate normalization of the terms followed by a syntactical check. One way to do so is to use hereditary substitutions as did Abel or Keller and Altenkirch in their respective formalizations[1, 16]. Another one uses the typing information available to build semantical objects where reductions *just happen*. This is our preferred choice.

[1] We used Agda extended with a postulate stating extensional equality for non-dependent functions in our formalization.

[2] On top of the traditional $f\ \text{`\$}\ x \rightsquigarrow_\eta (\text{`}\lambda\ .f\ \text{`\$}\ \text{`v}\ 0)\ \text{`\$}\ x \rightsquigarrow_\beta f\ \text{`\$}\ x$, the reader could think of e.g. $\text{`map}(f, xs) \rightsquigarrow_\eta \text{`map}(f, xs\ \text{`++}\ \text{`[]}) \rightsquigarrow_\eta \text{`map}(f, xs)\ \text{`++}\ \text{`map}(f, \text{`[]}) \rightsquigarrow_\iota \text{`map}(f, xs)\ \text{`++}\ \text{`[]}$ which is non-terminating *and* growing!

Type directed partial evaluation

Type directed partial evaluation is used as a way to get these 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[6] in order to have an efficient normalization procedure for LEGO. It has since been largely studied in different settings:

Danvy's lecture notes[13] 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[3] to get cleaner code generated.

T. Coquand and Dybjer[10] 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 naive 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 idea was the missing bit allowing us to get a model for weak-head normalization.

C. Coquand[9] 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 and last but not least Ahman[12] explained how to treat calculi equipped with algebraic effects.

[3] 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.

Reduction to weak-head normal forms rather than normal forms is the execution model of lazy languages (e.g. Haskell) and allows to deal with infinite objects and potential non termination quite liberally. It is also used in total languages such as Coq in order to have a cheap convertibility check: two elements with different head constructor are non convertible no matter what the evaluation of their respective subterms is.

Definition A term in weak-head form is either a term with a constructor in head position (weak-head *normal* form) or the elimination of a variable (weak-head *neutral* form). No restrictions are imposed unto the shape of the rest of the term.

$$\frac{pr : \sigma \in \Gamma}{\text{`v}\, pr : \Gamma \vdash_{whne} \sigma} \qquad \frac{\mathtt{f} : \Gamma \vdash_{whne} \sigma \text{ `}\to \tau \qquad \mathtt{x} : \Gamma \vdash \sigma}{\mathtt{f}\text{ `\$ }\mathtt{x} : \Gamma \vdash_{whne} \sigma}$$

$$\frac{t : \Gamma \vdash_{whne} \sigma \text{ `}\times\ \tau}{\text{`}\pi_1\, t : \Gamma \vdash_{whne} \sigma} \qquad \frac{f : \Gamma \vdash \sigma \text{ `}\to \tau \qquad xs : \Gamma \vdash_{whne} \text{`}\mathtt{list}\ \sigma}{\text{`}\mathtt{map}(f, xs) : \Gamma \vdash_{whne} \text{`}\mathtt{list}\ \tau}$$

$$\frac{t : \Gamma \vdash_{whne} \sigma \text{ `}\times\ \tau}{\text{`}\pi_2\, t : \Gamma \vdash_{whne} \tau} \qquad \frac{xs : \Gamma \vdash_{whne} \text{`}\mathtt{list}\ \sigma \qquad ys : \Gamma \vdash \text{`}\mathtt{list}\ \sigma}{xs\ \text{`}{+}{+}\ ys : \Gamma \vdash_{whne} \text{`}\mathtt{list}\ \sigma}$$

$$\frac{c : \Gamma \vdash \sigma \text{ `}\to \tau \text{ `}\to \tau \qquad n : \Gamma \vdash \tau \qquad xs : \Gamma \vdash_{whne} \text{`}\mathtt{list}\ \sigma}{\text{`}\mathtt{fold}(c, n, xs) : \Gamma \vdash_{whne} \tau}$$

Table 2.1: Inductive definition of the weak-head neutral forms

$$\frac{t : \Gamma \vdash_{whne} \sigma}{\text{`}\uparrow t : \Gamma \vdash_{whnf} \sigma} \qquad \frac{t : \Gamma \cdot \sigma \vdash \sigma}{\text{`}\lambda t : \Gamma \vdash_{whnf} \sigma \text{ `}\to \tau}$$

$$\frac{}{\text{`}\langle\rangle : \Gamma \vdash_{whnf} \text{`}1} \qquad \frac{a : \Gamma \vdash \sigma \qquad b : \Gamma \vdash \tau}{a\ \text{`},\ b : \Gamma \vdash_{whnf} \sigma \text{ `}\times\ \tau}$$

$$\frac{}{\text{`}[] : \Gamma \vdash_{whnf} \text{`}\mathtt{list}\ \sigma} \qquad \frac{hd : \Gamma \vdash \sigma \qquad tl : \Gamma \vdash \text{`}\mathtt{list}\ \sigma}{hd\ \text{`}::\ tl : \Gamma \vdash_{whnf} \text{`}\mathtt{list}\ \sigma}$$

Table 2.2: Inductive definition of the weak-head normal forms

Each $\Gamma \vdash_{whne} \sigma$ or $\Gamma \vdash_{whnf} \sigma$ trivially corresponds to a term $\Gamma \vdash \sigma$. We'll either leave these coercions implicit or talk about the erasure of a weak-head term. The weakenings induced by context inclusions will be either left implicit or written wk_{inc} where inc is the inclusion proof when judged necessary.

2.1 Defining a model

Unlike full normalization, weak-head normalization is very adamant about never doing more than is needed: whenever possible, the procedure will behave just like the identity function e.g. when building the spine of arguments

applied to a variable. As a consequence, elements of the model built need to have a mean to mention their origins. The adopted solution is inspired by Coquand and Dybjer's glueing[10] allowing the algorithm to forget about reductions which were not supposed to happen

The structure of this model definition highlights the semantical nature of the process: at the end of the day, reductions are all about explaining how to eliminate weak-head normal forms.

Definition $\langle . \ , \ . \rangle\!\!\rightarrow .$ is a predicate transformer used to define the model: it limits recourses to live objects in places where the runtime behavior of the term may reveal useful. We call this operator a gluer.

$$\langle \Gamma \ , \ \sigma \rangle\!\!\rightarrow R = \Gamma \vdash \sigma \times \biguplus \begin{array}{l} \Gamma \vdash_{whne} \sigma \\ \Gamma \vdash_{whnf} \sigma \times R \end{array}$$

Definition The weak-head model $\mathcal{M}_{wh}$ is defined as a glueing on a mutually defined acting model $\mathcal{M}^{\star}_{wh}$ essentially explaining how to eliminate terms. $\mathcal{M}_{wh}$'s definition is parametric in the type index: it contains a well-typed term (the one the normalization procedure started from) packed with either a weak-head neutral or a weak-head normal together with a semantical object explaining how to compute with it.

$$\mathcal{M}_{wh}(\Gamma, \sigma) = \langle \Gamma \ , \ \sigma \rangle\!\!\rightarrow \mathcal{M}^{\star}_{wh}(\Gamma, \sigma)$$

On the other hand, the elements of the acting model $\mathcal{M}^{\star}_{wh}$ are defined by induction on the target type. Quite unsurprisingly semantical units and semantical inhabitants of base types do not contain any information[1], semantical pairs are pairs, semantical functions are functions, etc.

$$\begin{array}{lcl}
\mathcal{M}^{\star}_{wh}(\Gamma, \ . \ ) & : & \mathtt{type}_n \to \mathtt{Set} \\
\mathcal{M}^{\star}_{wh}(\Gamma, \ \sigma \ ) & \Leftarrow & \mathtt{type\text{-}induction}(\sigma) \\
\mathcal{M}^{\star}_{wh}(\Gamma, \ \text{`}\mathtt{1} \ ) & = & \top \\
\mathcal{M}^{\star}_{wh}(\Gamma, \ \text{`}\mathtt{b} \ k \ ) & = & \top \\
\mathcal{M}^{\star}_{wh}(\Gamma, \ \sigma \ \text{`}\times \ \tau \ ) & = & \mathcal{M}_{wh}(\Gamma, \sigma) \times \mathcal{M}_{wh}(\Gamma, \tau) \\
\mathcal{M}^{\star}_{wh}(\Gamma, \ \sigma \ \text{`}\rightarrow \ \tau \ ) & = & \forall \Delta \supseteq \Gamma, \mathcal{M}_{wh}(\Delta, \sigma) \to \mathcal{M}_{wh}(\Delta, \tau) \\
\mathcal{M}^{\star}_{wh}(\Gamma, \ \text{`}\mathtt{list} \ \sigma \ ) & = & \mathcal{L}^{\star}_{wh}(\Gamma, \sigma, \lambda\Gamma \to \mathcal{M}_{wh}(\Gamma, \sigma))
\end{array}$$

where $\mathcal{L}^{\star}_{wh}$ is an inductive set describing semantical lists; it is basically a list of semantical objects where the tail of a cons is always encapsulated in a gluer.

$$\frac{\Gamma : \mathtt{Con}(\mathtt{type}_n) \qquad \sigma : \mathtt{type}_n}{\Vdash_{\sigma} : \mathtt{Con}(\mathtt{type}_n) \to \mathtt{Set}} \qquad \frac{}{\text{`}[] : \mathcal{L}^{\star}_{wh}(\Gamma, \sigma, \Vdash_{\sigma})}$$

$$\frac{HD : \Vdash_{\sigma} (\Gamma) \qquad TL : \langle \Gamma \ , \ \text{`}\mathtt{list} \ \sigma \rangle\!\!\rightarrow \mathcal{L}^{\star}_{wh}(\Gamma, \sigma, \Vdash_{\sigma})}{HD \ \text{`}:: \ TL : \mathcal{L}^{\star}_{wh}(\Gamma, \sigma, \Vdash_{\sigma})}$$

[1] It is indeed unsurprising: the calculus is completely silent on the matter of base types so the only way to build inhabitants of one such type is through the use of variables hence the creation of weak-head neutrals. Quite straightforwardly, there is no need for an explanation on how to eliminate weak-head normals of base type.

It should be noted that this inductive definition is parametric in its arguments which guarantees that there is no positivity problem when building the model.

Remark For a given type σ the models behave like functors for the category of contexts and inclusions. In other words: weakenings do exist and the appropriate identities hold.

Lemma 2.1.1 (Reify and reflect)**.** *It is possible to extract weak-head normals* $\Gamma \vdash_{whnf} \sigma$ *from the model* $\mathcal{M}_{wh}(\Gamma, \sigma)$ *as well as inject weak-head neutrals* $\Gamma \vdash_{whne} \sigma$ *in it.*

Proof. From this model, it is trivial to extract a weak-head normal form: the glueing provides either a term already in weak-head normal form or a weak-head neutral which we can lift using the $\text{`}\uparrow$ constructor. We note $\uparrow_\sigma$ the type-indexed[2] function performing this operation.

It is as simple to inject a weak-head neutral t into the model: the first component will be the erasure of t and the second will simply be the left injection of t. We note $\downarrow_\sigma$ the corresponding type-indexed function[2]. □

These model definitions (and the corresponding weakenings) can be naturally extended pointwise to pairs of contexts. This give rise to the notion of semantical environments noted $\mathcal{M}\varepsilon_{wh}$. Thanks to $\downarrow_\sigma$ and environment weakening, one can define the canonical inhabitant of the diagonal environment $\mathcal{M}\varepsilon_{wh}(\Gamma, \Gamma)$ by a trivial induction on Γ.

2.2 Interpreting terms in this model

Once the model is set up, it is left to prove that terms can be immersed in the model were computations will happen before reification brings back normal forms. In order not to go crazy when proving soundness later on, it is wise to give this evaluation function as much structure as possible. This is done through a multitude of intermediate lemmas explaining the semantical counterparts of the usual notions of the calculus.

Let's walk through the definition of the interpretation of map over lists in the model. Variable lookup, projections of a tuples' components, fold, etc. can all be given semantical counterparts following the scheme described here without any particular difficulty.

Remark The semantical counterpart of a term of the calculus will be using the same name except that it will be suffixed with a prime e.g. the interpretation of $\text{`}\texttt{map}$ is $\text{`}\texttt{map}'$.

Lemma 2.2.1 (Semantical application)**.** $\mathcal{M}_{wh}$ *is closed under application; i.e. from every pair* $\mathcal{M}_{wh}(\Gamma, \sigma \text{`}\to \tau) \times \mathcal{M}_{wh}(\Gamma, \sigma)$*, one can build an object of* $\mathcal{M}_{wh}(\Gamma, \tau)$*.*

[2] These functions are not really type-indexed but their counterpart for the full-normalization algorithm are. In order to have coherent notations, we stick to mentioning the type.

Proof. Let (f, F) and (x, X) respectively be the objects in $\mathcal{M}_{wh}(\Gamma, \sigma \text{ `}\!\!\rightarrow \tau)$ and $\mathcal{M}_{wh}(\Gamma, \sigma)$. Because the first component of $\mathcal{M}_{wh}$ should always be the term the reduction started from, it will be $f \text{ `}\!\$ \, x$. The second component is built by case analysis on F.

If F is the left injection of some weak-head neutral f_{whne} then $f_{whne} \text{ `}\!\$ \, x$ also is a weak-head neutral hence $\iota_1(f_{whne} \text{ `}\!\$ \, x)$ is of the appropriate type.

If F is the right injection of a pair $(f_{whnf}, F^\star)$ then $F^\star$ is an object of $\mathcal{M}^\star_{wh}(\Gamma, \sigma \text{ `}\!\!\rightarrow \tau)$ i.e. a function producing elements of $\mathcal{M}_{wh}(\_, \tau)$. Returning the second projection of $F^\star(\mathtt{id}_\subseteq(\Gamma), (x, X))$ concludes the proof. □

Lemma 2.2.2 (Semantical "cons")**.** *There is a concept similar to cons for lists in the model: given a head* $\mathcal{M}_{wh}(\Gamma, \sigma)$ *and a tail* $\mathcal{M}_{wh}(\Gamma, \text{`}\mathtt{list}\ \sigma)$*, one can build a list* $\mathcal{M}_{wh}(\Gamma, \text{`}\mathtt{list}\ \sigma)$*.*

Proof. Let (hd, HD) be the head and (tl, TL) be the tail. The term $t = hd \text{ `}\!:\!: \, tl$ is a weak-head normal $\Gamma \vdash_{whnf} \text{`}\mathtt{list}\ \sigma$ and $T = (hd, HD) \text{ `}\!:\!: (tl, TL)$ is an element of the action model $\mathcal{M}^\star_{wh}(\Gamma, \text{`}\mathtt{list}\ \sigma)$ whence the well-typed semantical object corresponding to the *consing* of the head on the tail: $(t, \iota_2(t, T))$. □

Lemma 2.2.3 (Semantical map)**.** *Given a semantical functional* $\mathcal{M}_{wh}(\Gamma, \sigma \text{ `}\!\!\rightarrow \tau)$*, one can define its mapping over a semantical list* $\mathcal{M}_{wh}(\Gamma, \text{`}\mathtt{list}\ \sigma)$ *outputing another semantical list* $\mathcal{M}_{wh}(\Gamma, \text{`}\mathtt{list}\ \tau)$*.*

Proof. Let (f, F) be the functional and (xs, XS) be the list. The first component of the returned $\mathcal{M}_{wh}(\Gamma, \text{`}\mathtt{list}\ \tau)$ will be $\text{`}\mathtt{map}(f, xs)$. The second one is defined by case analysis on XS.

If XS is the left injection of some weak-head neutral xs_{whne} then the stuck map $\text{`}\mathtt{map}(f, xs_{whne})$ is a neutral and its left injection is of the expected type.

If XS is the right injection of a pair $(xs_{whnf}, XS^\star)$ then we proceed by induction on the semantical object $XS^\star$ of type $\mathcal{M}^\star_{wh}(\Gamma, \text{`}\mathtt{list}\ \sigma)$.

- If $XS^\star$ is the empty list then mapping F over it should just yield the empty list
- If $XS^\star$ is $HD \text{ `}\!:\!:' \, TL$ then the induction hypothesis provides $YS$, the result of $\text{`}\mathtt{map}'((f, F), TL)$. Combining it with $(f, F) \text{ `}\!\$' \, HD$ using the semantical cons $\_ \text{ `}\!:\!:' \_$ builds just the needed term[3].

□

After describing how the expected behaviour of each term constructor can be explained in terms of actions on the elements of the model, all these can be combined in order to show that terms can be embedded in $\mathcal{M}_{wh}$ given an embedding of their free variables.

Theorem 2.2.4 (Evaluation function)**.** *Given a term* t *of type* $\Gamma \vdash \sigma$ *and a semantical environment* $\mathcal{M}\varepsilon_{wh}(\Delta, \Gamma)$*, there exists a semantical object* T *of type* $\mathcal{M}_{wh}(\Delta, \sigma)$*.*

[3] Up-to an additional $\Gamma \vdash \text{`}\mathtt{list}\ \tau$ which is discarded by taking the second projection of the result.

Proof. Let R be the semantical environment $\mathcal{M}\varepsilon_{wh}(\Delta, \Gamma)$. The proof is done by structural induction on t using either the semantical operations described earlier in order to combine the induction hypothesis or looking up a term in the environment in the variable case. The only case left is the lambda abstraction one.

$\text{`}\lambda\, t$ is a weak-head normal form thus the need for an $\mathcal{M}^{\star}_{wh}(\Gamma, \sigma \text{ `}\!\!\rightarrow \tau)$ object T to build the $\mathcal{M}_{wh}(\Gamma, \sigma \text{ `}\!\!\rightarrow \tau)$ element $(\text{`}\lambda\, t, \iota_2(\text{`}\lambda\, t, T))$. Given E a context, *inc* an inclusion proof $\Delta \subseteq E$ and X a semantical object $\mathcal{M}_{wh}(E, \sigma)$, one has to build a $\mathcal{M}_{wh}(E, \tau)$ element. By combining X with an *inc* weakening of R, one can get a semantical environment $\mathcal{M}\varepsilon_{wh}(E, \Gamma \cdot \sigma)$ which, together with t, produces the required element by induction hypothesis. □

Corollary 2.2.5 (Weak-head normalization)**.** *From each term of type* $\Gamma \vdash \sigma$*, one can derive a weak-head normal term* $\Gamma \vdash_{whnf} \sigma$*.*

Proof. By combining the evaluation function, the existence of diagonal elements in $\mathcal{M}\varepsilon_{wh}$ and the weak-head normal forms extractor $\uparrow_{\sigma}$, one can immerse t in $\mathcal{M}_{wh}(\Gamma, \sigma)$ and then extract a corresponding $\Gamma \vdash_{whnf} \sigma$. This weak-head normalization function is called `wh-norm`. □

2.3 Reassuring oneself

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

Its role is to formalize the intuition mentionned when building $\mathcal{M}_{wh}$: the first component is the term the reduction procedure started from and the elements of the active model are reducts of the weak-head normal they are glued to. The first result can be proved by a simple induction on t.

Lemma 2.3.1. *The first component of the evaluation of* t *in the environment* R *is* t *where the substitution* ρ pulled[4] *from* R *has been applied.*

Proof. By induction on t using trivial lemmas for each semantical operator described earlier on. □

In order to prove the second part of the informal specification, a logical relation is needed: knowledge about the behaviour of a function is acquired when evaluating a lamda abstraction but is used when applying it ; the same goes for e.g. pairings and projections. Given that the active model is only used in the presence of weak-head normals, the relation will connect such weak-head normals to the live objects. The reader will therefore understand that it is preferable to start by explaining once and for all how to lift such a relation through a gluer.

[4]The first component of a semantical object $\mathcal{M}_{wh}(\Gamma, \sigma)$ is a term $\Gamma \vdash \sigma$ thence the possibility to build up a substitution $\Delta \vdash\!\varepsilon\, \Gamma$ from a semantical environment $\mathcal{M}\varepsilon_{wh}(\Delta, \Gamma)$ by taking the first projection of all of its elements.

Definition Given a logical relation $\mathcal{R}_{\Gamma,\sigma}$ relating weak-head normal forms $\Gamma \vdash_{whnf} \sigma$ to elements of R, one can lift it along the gluer in the following fashion: let (t, T) be an element of $\langle \Gamma \ , \ \sigma \rangle\!\!\supset R$.

if T is the left injection of a weak head neutral element t_{whne} then the requirement is $t \rightsquigarrow^*_{\beta\eta\iota} t_{whne}$;

if T is the right injection of a pair $(t_{whnf}, T^\star)$ containing a weak head normal term and an element of R then the requirement is that $t \rightsquigarrow^*_{\beta\eta\iota} t_{whnf}$ and $\mathcal{R}_{\Gamma,\sigma}(t_{whnf}, T^\star)$.

Definition The structure of the logical relation (denoted $\lightning_{wh}(.)(.)$) follows closely the model's one. It is defined on $\mathcal{M}_{wh}$ as the lifting along the gluer of the one defined on the acting model $\mathcal{M}^\star_{wh}$ which, itself, is built by induction on the index type σ:

$$\begin{array}{lll} \mathcal{M}^\star_{wh}(\Gamma,\sigma)\ .\ \ni\ .\ \lightning^\star_{wh} & : & \Gamma \vdash_{whnf} \sigma \to \mathcal{M}^\star_{wh}(\Gamma,\sigma) \to \texttt{Set} \\ \mathcal{M}^\star_{wh}(\Gamma,\sigma)\ t\ \ni\ T\ \lightning^\star_{wh} & \Leftarrow & \texttt{type-induction}(\sigma) \end{array}$$

In the unit as well as the base type case, there is no need for an information transfer given that for these types all the semantical objects in the active model $\mathcal{M}^\star_{wh}(\Gamma, .)$ are identical. Our only requirement is therefore a proof of the trivial proposition:

$$\begin{array}{lll} \mathcal{M}^\star_{wh}(\Gamma, \text{`}\texttt{1}) \ni \_ \lightning^\star_{wh} \_ & = & \top \\ \mathcal{M}^\star_{wh}(\Gamma, \text{`}\texttt{b}\ k) \ni \_ \lightning^\star_{wh} \_ & = & \top \end{array}$$

In the product case, the semantical object is actually a pair (A, B) of semantical objects corresponding to the left and the right components of the product. One should require that the weak-head normal form reduces to the pairing of the source elements of these two semantical objects and that this two semantical objects are themselves well-behaved.

$$\mathcal{M}^\star_{wh}(\Gamma, \sigma\ \text{`}\times\ \tau) \ni p\ \lightning^\star_{wh}\ A, B = p \rightsquigarrow^*_{\beta\eta\iota} (\pi_1\ A, \pi_1\ B) \times \lightning_{wh}(\sigma)(A) \times \lightning_{wh}(\tau)(B)$$

In the function case, one expects every well-behaved input to be turned into a well-behaved output. In addition, one should mention that the first component of the output could be anything *above* (in terms of the reduction relation) the simple application of terms.

$$\begin{array}{l} \mathcal{M}^\star_{wh}(\Gamma, \sigma\ \text{`}\to\ \tau) \ni f\ \lightning^\star_{wh}\ F = \\ \forall(\Delta : \texttt{Con}(\texttt{type}_n))(inc\colon \Delta \supseteq \Gamma)(X : \mathcal{M}_{wh}(\Delta, \sigma))(X\lightning : \lightning_{wh}(\sigma)(X)) \\ (y : \Delta \vdash \tau) \to y \rightsquigarrow^*_{\beta\eta\iota} f\ \text{`}\$\ \pi_1\ X \to \lightning_{wh}(\tau)(y, \pi_2(F(inc, X))) \end{array}$$

Finally the list case is handled by an auxiliary definition, once again, parametric in its arguments.

$$\mathcal{M}^\star_{wh}(\Gamma, \text{`}\texttt{list}\ \sigma) \ni xs\ \lightning^\star_{wh}\ XS = \mathcal{L}^\star_{wh}(\Gamma, \sigma, \Vdash_\sigma)\ \ni\ xs\ \lightning\ XS\ \textit{by}\ (\lightning_{wh}(\sigma)(.))$$

Given a predicate $\Vdash_\sigma \lightning_{wh}\colon \forall\Gamma, \Vdash_\sigma(\Gamma) \to \texttt{Set}$ morally caracterizing the elements of $\Vdash_\sigma(\Gamma)$ that are well-behaved, one can build the logical relation relating a weak-head normal term xs_{whnf} of type $\Gamma \vdash_{whnf} \text{`}\texttt{list}\ \sigma$ and a semantical list XS of type $\mathcal{L}^\star_{wh}(\Gamma, \sigma, \Vdash_\sigma)$ by induction on XS.

if XS is the empty list then the requirement is that the weak-head normal form computes down to the empty list.

if XS is a non empty list $HD \mathbin{`::'} TL$ then one expects the weak-head normal form to reduce to the non empty list built from the source elements of HD and TL. On top of that, HD should be well-behaved[5] and the gluer's extension of the induction hypothesis for TL should hold.

The first thing to notice is that proving that the evaluation function produces terms which agree with the logical relation is enough to prove soundness. The reification of a term for which the logical relation holds is indeed a reduct of its first component:

Proposition 2.3.2. *$\lightning_{wh}(\sigma)(t,T)$ implies that $t \rightsquigarrow^*_{\beta\eta\iota} \uparrow_\sigma(t,T)$.*

Proof. Trivial by case analysis on T. □

The evaluation function is structured around combining the semantical counterparts of the term constructors together with semantical objects obtained *via* induction hypothesis. It should therefore be possible to build the soundness proof in a modular fashion: if every semantical construct preserves the logical relation's validity then the whole evaluation process must be sensible.

One of the common practices in the definition of these semantical operators is to drop the first component of an object to replace it with *the term we really started from*. In other words: if the validity of the logical relations is to remain true throught the use of these operators, it has to be upward-closed under the reduction relation.

Lemma 2.3.3 (Closures)**.** *$\mathcal{M}^\star_{wh}(\Gamma,\sigma) \ni . \lightning^\star_{wh} T$ and $\lightning_{wh}(\sigma)(.,T)$ are upward-closed with respect to the reduction relation.*

Proof. This can be proved by a simple case analysis on σ followed by a case analysis on T in the list case for $\mathcal{M}^\star_{wh}(\Gamma,\sigma) \ni . \lightning^\star_{wh} T$ and a simple case analysis on T for $\lightning_{wh}(\sigma)(.,T)$. □

Let's walk through the proof that map is indeed correct. We call well-behaved the elements T of the model $\mathcal{M}_{wh}(\Gamma,\sigma)$ for which $\lightning_{wh}(\sigma)(T)$ holds. The proofs are all done by functional induction i.e. by following the way the semantical operator was defined.

Lemma 2.3.4 (Semantical application)**.** *Given a well-behaved function in $\mathcal{M}_{wh}(\Gamma,\sigma \mathbin{`\to} \tau)$ and a well-behaved argument $\mathcal{M}_{wh}(\Gamma,\sigma)$, their semantical application is also well-behaved.*

Proof. Let (f,F) be the well-behaved functional and (x,X) be the corresponding argument. The functional induction on the semantical application distinguishes two cases:

Either F is the left injection of some weak-head neutral f_{whne} where $f \rightsquigarrow^*_{\beta\eta\iota} f_{whne}$ holds by hypothesis. The proof requirement for the corresponding semantical application is $f \mathbin{`\$} x \rightsquigarrow^*_{\beta\eta\iota} f_{whne} \mathbin{`\$} x$ which is trivially discharged by structural lifting of the hypothesis.

Or F is the right injection of the pairing of a weak-head normal form f_{whnf} and an element of the action model $F^\star$. $f \mathbin{`\$} x \rightsquigarrow^*_{\beta\eta\iota} f_{whnf} \mathbin{`\$} x$ trivially

[5] Meaning that $\Vdash_{\mathscr{A}} \lightning_{wh}(\Gamma, HD)$ should hold.

holds because of the structural rule for application and one of the hypothesis given by the fact that (f, F) is well-behaved. Combined with the fact that $\mathcal{M}^{\star}_{wh}(\Gamma, \sigma \text{ `}\!\!\to \tau) \ni f_{whnf} \not\downarrow^{\star}_{wh} F$ holds and that (x, X) is well-behaved this concludes the proof. □

Lemma 2.3.5 (Semantical cons)**.** *The semantical consing of a well-behaved head and tail yields a well-behaved list.*

Proof. Let (hd, HD) be the head and (tl, TL) be the tail. The proof is done by case analysis on TL: in both cases the things to prove are either trivial or assumptions. □

Lemma 2.3.6 (Semantical map)**.** *The semantical map preserves the property of being well-behaved.*

Proof. Let (f, F) be the function and (xs, XS) the list. The functional induction distinguishes three cases.

Either XS is the left injection of a stuck term xs_{whne} and proving the goal amounts to combining the assumption about (xs, XS) and the structural rule for the second argument of `map.

Or XS is the right injection of an xs_{whnf} and the empty semantical list. Then by assumption $xs \rightsquigarrow^{*}_{\beta\eta\iota} xs_{whnf}$ and $xs_{whnf} \rightsquigarrow^{*}_{\beta\eta\iota}$ `[] hence by structural rules and $\iota$ for map, `map$(f, xs) \rightsquigarrow^{*}_{\beta\eta\iota}$ `[].

Or XS is the right injection of an xs_{whnf} and the semantical cons HD `:: $'$ $TL$. By induction hypothesis, `map$'((f, F), TL)$ is well-behaved and by lemma 2.3.4, (f, F) `\$$'$ $HD$ is well behaved. Hence $((f, F)$ `\$$'$ $HD)$ `:: $'$ `map$'((f, F), TL)$ is also well-behaved. Finally because `map$(f, xs) \rightsquigarrow^{*}_{\beta\eta\iota} (f$ `\$ π_1 $HD)$ `:: `map$(f, \pi_1\, TL)$ and being well-behaved is closed under $\beta\eta\iota$ expansion of the first term, one can finish the proof. □

Theorem 2.3.7. *The evaluation function is well-behaved provided that the semantical environment used is.*

Proof. Let t be the term and R be the well-behaved environment. All the cases can be dealt with by fitting together inductions hypotheses and the auxiliary lemmas proved earlier except for the lambda abstraction. In the case of the lambda abstraction $t =$ `$\lambda\, b$, the induction hypothesis ensures that for $\Delta$ an extension of $\Gamma$ and $X$ well-behaved in $\mathcal{M}(\Delta, \sigma)$, `eval$(b, (R, X))$ is well-behaved.

Recall that the first component of an evaluation is the evaluated term where free variables have been substituted by terms pulled from the semantical environment. Let's call ρ the substitution pulled from R whence the first component of `eval$(b, (R, X))$ is $b[\rho, \pi_1\, X]$ which is a direct reduct of (`$\lambda\, b)[\rho]$ `\$$(\pi_1\, X)$ thus the conclusion of the proof by upward closure of the logical relation. □

Corollary 2.3.8 (Soundness)**.** *For all term t of type $\Gamma \vdash \sigma$, t reduces by $\beta\iota$ to the erasure of* `wh-norm` *t.*

Proof. $\downarrow_{\sigma}$ quite trivially produces well-behaved elements of the model ; moreover semantical weakening is compatible with the logical relation thence the diagonal semantical environment is well-behaved and, by the preceding theorem, `eval$(t,$ `diag$(\Gamma))$ is well-behaved.

As a consequence, lemma 2.3.2 and the fact that $\texttt{pull}(\texttt{diag}(\Gamma))$ is the identity substitution ensure that $t \rightsquigarrow^*_{\beta\eta\iota} \texttt{wh-norm}\, t$. □

Normalization and standardization of lambda terms can be of different interests, our original motivation is to get a more liberal definitional equality relieveing the mathematician's shoulder of at least part of the proof burden they have to endure when certifying a program correct. But it can also be used as a compilation technique to compute away expressions which are statically known (as explained by e.g. Danvy[13]) or optimize list traversals by fusioning different steps (as presented by e.g. Hinze et al.[15]).

The more liberal definitional equality we want shoudl still be decidable given that, when extending the calculus to dependent types, such a test will be required by any typechecker. In order to be able to decide equality up-to the equational theory, one has to have a mean to extract canonical representatives of the equivalence classes. These representatives are the normal and neutral forms described here and they are picked thanks to a normalization by evaluation algorithm.

The soundness and completeness proofs ensure that the representatives are unique and that the algorithm never pairs up a term with the representative of a different class.

Definition Neutral and normal forms are mutually defined so that they cannot possibly contain β or ι redexes: neutrals represent stuck eliminations while normals specify canonical shapes for terms depending on the type they have.

$$\frac{pr : \sigma \in \Gamma}{\text{`v}\, pr : \Gamma \vdash_{ne} \sigma} \qquad \frac{\mathtt{f} : \Gamma \vdash_{ne} \sigma \text{ `}\to \tau \qquad \mathtt{x} : \Gamma \vdash_{nf} \sigma}{\mathtt{f}\ \text{`\$}\ \mathtt{x} : \Gamma \vdash_{ne} \sigma}$$

$$\frac{t : \Gamma \vdash_{ne} \sigma \text{ `}\times\ \tau}{\text{`}\pi_1\, t : \Gamma \vdash_{ne} \sigma} \qquad \frac{t : \Gamma \vdash_{ne} \sigma \text{ `}\times\ \tau}{\text{`}\pi_2\, t : \Gamma \vdash_{ne} \tau}$$

$$\frac{c : \Gamma \vdash_{nf} \sigma \text{ `}\to \tau \text{ `}\to \tau \qquad n : \Gamma \vdash_{nf} \tau \qquad xs : \Gamma \vdash_{ne} \text{`}\mathtt{list}\ \sigma}{\text{`}\mathtt{fold}(c, n, xs) : \Gamma \vdash_{ne} \tau}$$

Table 3.1: Inductive definition of the neutral forms

$$\frac{t : \Gamma \vdash_{ne} \text{`}\mathtt{b}\ k}{\text{`}\uparrow t : \Gamma \vdash_{nf} \text{`}\mathtt{b}\ k} \qquad \frac{t : \Gamma \cdot \sigma \vdash_{nf} \sigma}{\text{`}\lambda\, t : \Gamma \vdash_{nf} \sigma \text{ `}\to \tau}$$

$$\frac{}{\text{`}\langle\rangle : \Gamma \vdash_{nf} \text{`}\mathtt{1}} \qquad \frac{a : \Gamma \vdash_{nf} \sigma \qquad b : \Gamma \vdash_{nf} \tau}{a\ \text{`},\ b : \Gamma \vdash_{nf} \sigma \text{ `}\times\ \tau}$$

$$\frac{}{\text{`}[] : \Gamma \vdash_{nf} \text{`}\mathtt{list}\ \sigma} \qquad \frac{hd : \Gamma \vdash_{nf} \sigma \qquad tl : \Gamma \vdash_{nf} \text{`}\mathtt{list}\ \sigma}{hd\ \text{`::}\ tl : \Gamma \vdash_{nf} \text{`}\mathtt{list}\ \sigma}$$

$$\frac{f : \Gamma \vdash_{nf} \sigma \text{ `}\to \tau \qquad xs : \Gamma \vdash_{ne} \text{`}\mathtt{list}\ \sigma \qquad ys : \Gamma \vdash_{nf} \text{`}\mathtt{list}\ \tau}{\mathtt{mappend}(f, xs, ys) : \Gamma \vdash_{nf} \text{`}\mathtt{list}\ \tau}$$

Table 3.2: Inductive definition of the normal forms

Each $\Gamma \vdash_{ne} \sigma$ or $\Gamma \vdash_{nf} \sigma$ trivially corresponds to a term $\Gamma \vdash \sigma$. We'll either leave these coercions implicit or talk about the erasure of a weak-head term. The weakenings induced by context inclusions will be either left implicit or written wk_{inc} where inc is the inclusion proof when judged necessary.

3.1 Computing

The model designed to achieve full normalization and standardization is both simpler and more complex than the one crafted to produce weak-head forms: a glueing is not required here given that no computation will ever be cancelled but there is need for more complex semantical lists. Indeed, standardization cares about stuck maps or appends and needs to somehow store their computational power and be ready to fire some more reductions when rewriting fusion rules creating new redexes. Hence the following definition by induction on the type.

Definition The calculus enjoys η rules both for unit, product and arrow type thence the absence of subtlety in the model definition: all the semantical units are units, the semantical pairs are pairs of semantical objects and the semantical functions are functions on semantical objects. The fact that base types do not have canonical constructors is also reflected by their semantical interpretation being plain neutrals.

$$\begin{array}{lcl}
\mathcal{M}(\Gamma,\ \sigma\ ) & \Leftarrow & \texttt{type-induction}(\sigma) \\
\mathcal{M}(\Gamma,\ \text{`}\texttt{1}\ ) & = & \top \\
\mathcal{M}(\Gamma,\ \text{`}\texttt{b}\ k\ ) & = & \Gamma \vdash_{ne} \text{`}\texttt{b}\ k \\
\mathcal{M}(\Gamma,\ \sigma\ \text{`}\times\ \tau\ ) & = & \mathcal{M}(\Gamma,\sigma) \times \mathcal{M}(\Gamma,\tau) \\
\mathcal{M}(\Gamma,\ \sigma\ \text{`}\rightarrow\ \tau\ ) & = & \forall \Delta\ \supseteq\ \Gamma, \mathcal{M}(\Delta,\sigma) \rightarrow \mathcal{M}(\Delta,\tau) \\
\mathcal{M}(\Gamma,\ \text{`}\texttt{list}\ \sigma\ ) & = & \mathcal{L}(\Gamma,\sigma,\lambda\Gamma \rightarrow \mathcal{M}(\Gamma,\sigma))
\end{array}$$

where $\mathcal{L}(\Gamma,\sigma,\Vdash_\sigma)$ is and inductive definition parametric in its arguments:

$$\frac{\Gamma : \texttt{Con}(\texttt{type}_n) \qquad \sigma : \texttt{type}_n \qquad \Vdash_\sigma : \texttt{Con}(\texttt{type}_n) \rightarrow \texttt{Set}}{\mathcal{L}(\Gamma,\sigma,\Vdash_\sigma) : \texttt{Set}}$$

$$\frac{}{\text{`}[] : \mathcal{L}(\Gamma,\sigma,\Vdash_\sigma)} \qquad \frac{HD :\ \Vdash_\sigma(\Gamma) \qquad TL : \mathcal{L}(\Gamma,\sigma,\Vdash_\sigma)]}{HD\ \text{`}::\ TL : \mathcal{L}(\Gamma,\sigma,\Vdash_\sigma)}$$

$$\frac{f : \forall \Delta \supseteq \Gamma, \Delta \vdash_{ne} \tau \rightarrow \Vdash_\sigma(\Delta) \qquad xs : \Gamma \vdash_{ne} \text{`}\texttt{list}\ \tau \qquad YS : \mathcal{L}(\Gamma,\sigma,\Vdash_\sigma)}{\texttt{mappend}(f,xs,YS) : \mathcal{L}(\Gamma,\sigma,\Vdash_\sigma)}$$

where $\texttt{mappend}(f,xs,YS)$ intuitively means $\text{`}\texttt{map}(f,xs)\ \text{`}{+}{+}\ YS$ for a stuck xs and in which only the vital information[1] about f has been stored.

Lemma 3.1.1 (Reify and reflect)**.** *Mutually defined processes allow normal forms $\Gamma \vdash_{nf} \sigma$ to be extracted from elements of the model $\mathcal{M}(\Gamma,\sigma)$ while neutral forms $\Gamma \vdash_{ne} \sigma$ can be turned into elements of the model.*

[1] The vital information being its behaviour on variables which is just enough to be able to compute compositions and to trigger the reification mecanism. Note that this voluntary limitation to neutrals in the domain is not the result of an ascetic whim; it is absolutely impossible to have a well-founded definition of the model and live objects in semantical stuck maps: τ is nowhere related to $\text{`}\texttt{list}\ \sigma$.

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

The unit case is trivial: the reification process returns $\text{`}\Diamond$ while the reflection one produces the only inhabitant of $\top$.

$$\uparrow_{\text{`1}\_} = \text{`}\Diamond \qquad\qquad \downarrow_{\text{`1}\_} = \texttt{tt}$$

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.

$$\uparrow_{\text{`b}\ k} t = \text{`}\uparrow t \qquad\qquad \downarrow_{\text{`b}\ k} t = t$$

The product case is solved by induction hypothesis: the reification is the pairing of the reification of the subterms while the reflection is the reflection of the η-expansion of the stuck term.

$$\uparrow_{\sigma \times \tau}(A, B) = (\uparrow_\sigma A, \uparrow_\tau B) \qquad\qquad \downarrow_{\sigma \times \tau} t = (\downarrow_\sigma \text{`}\pi_1\, t, \downarrow_\tau \text{`}\pi_2\, t)$$

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

$$\begin{array}{lcl} \uparrow_{\sigma \hookrightarrow \tau} F & = & \text{`}\lambda(\uparrow_\tau F(\texttt{step}(\texttt{id}_\subseteq), \downarrow_\sigma \text{`}\texttt{v}\,0)) \\ \downarrow_{\sigma \hookrightarrow \tau} f & = & \lambda\Delta\ inc\ x.\downarrow_\tau (wk_{inc}(f)\ \text{`\$}\uparrow_\sigma x) \end{array}$$

The list case is dealt with by recursion on the semantical list for the reification process and a simple injection for the reflection case:

$$\begin{array}{lcl} \uparrow_{\text{`list}\ \sigma} \quad \text{`}[] & = & \text{`}[] \\ \uparrow_{\text{`list}\ \sigma} \quad HD\ \text{`::}\ TL & = & (\uparrow_\sigma HD)\ \text{`::}\ (\uparrow_{\text{`list}\ \sigma} TL) \\ \uparrow_{\text{`list}\ \sigma}\ \texttt{mappend}(f, xs, YS) & = & \text{`}\texttt{map}(\text{`}\lambda(\uparrow_\sigma(f(\text{`}\texttt{v}\,0))), xs)\ \text{`}{+\!+}\ (\uparrow_{\text{`list}\ \sigma} YS) \end{array}$$

$$\downarrow_{\text{`list}\ \sigma} xs = \texttt{mappend}(\downarrow_\sigma, xs, \text{`}[])$$

□

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 $\mathcal{M}\varepsilon(\Delta, \Gamma)$. Indeed the definition of the reflection function $\downarrow_\sigma$ together with the existence of environment weakenings give us the necessary machinery to produce a diagonal semantical environment $\mathcal{M}\varepsilon(\Gamma, \Gamma)$ which could then be fed to such an evaluation function.

Unlike the involved construction needed for weak-head normalization, this model is quite straightforward to inhabit. The reader can refer to Dybjer[10] or Coquand[9] if she wants to have an idea of how to interpret the usual constructors while we will focus on the original content of this paper: lists and their special reduction rules.

Lemma 3.1.2 (Semantical map)**.** *From a semantical functional in* $\mathcal{M}(\Gamma, \sigma \text{`}\to \tau)$ *and a semantical list in* $\mathcal{M}(\Gamma, \text{`}\texttt{list}\ \sigma)$*, one can build a semantical list in* $\mathcal{M}(\Gamma, \text{`}\texttt{list}\ \tau)$*.*

Proof. Let F be the functional and XS be the list; the mapping of F on XS is defined by induction on XS:

- If XS is empty, so is the output;

- If XS is HD `:: $'TL$, one applies $F$ to $HD$ and produces a tail corresponding to `map$'(F, TL)$ by induction hypothesis;
- If XS is a stuck map-append `mappend(G, xs_{ne}, TL) then the result is also a stuck map-append combining the composition of F with G and the induction hypothesis:

  `mappend$(\lambda inc.\lambda x_{ne}.F(inc, G(inc, x_{ne})), xs_{ne},$ `map$'(F, TL))$

□

Lemma 3.1.3 (Semantical append)**.** *From two semantical lists in* $\mathcal{M}(\Gamma,$ `list $\sigma)$, *one can build the list corresponding to appending the second one at the end of the first one.*

Proof. Let XS and YS be the two semantical lists, the result is built by induction on XS.

- If XS is empty then appending YS at its end amounts to returning YS;
- If XS is equal to HD `:: $'TL$ then $TL$ `++ $'YS$ is given by induction hypothesis and the output is HD `:: $'(TL$ `++ $'YS)$;
- If XS is `mappend$(F, xs_{ne}, TL)$ then the output is combining the induction hypothesis giving $TL$ `++ $'YS$ in a way that exploits associativity of append: `mappend$(F, xs_{ne}, TL$ `++ $'YS)$

□

Lemma 3.1.4 (Semantical fold)**.** *One can build a semantical counterpart to the fold function.*

Proof. Let C and N respectively be the elements in $\mathcal{M}(\Gamma, \sigma$ `→ $\tau$ `→ $\tau)$ and $\mathcal{M}(\Gamma, \tau)$ and XS be the semantical list in $\mathcal{M}(\Gamma,$ `list $\sigma)$. As usual the output is defined by induction on XS.

- If XS is empty, one returns N;
- If XS is HD `:: $'TL$, one returns $C(HD,$ `fold$'(C, N, TL))$;
- If XS is a stuck `mappend(F, xs_{ne}, TL) then one has to perform fold-append fusion as well as fold-map fusion. Given that xs_{ne} is a neutral, the only way to actually produce a $\mathcal{M}(\Gamma, \tau)$ is through reflection of a neutral. Hence the following construction:

  First of all, build $c' : \Gamma \vdash_{nf} \sigma$ `→ $\tau$ `→ τ the reification of the composition of C and F: `λ `λ$(\uparrow_\tau C($`pr`$, F($`pr`$,$ `v `1$)$, `id$_\subseteq(\Gamma), \downarrow_\tau$ `v `0$))$ where `pr is the trivial proof of $\Gamma \subseteq \Gamma \cdot \sigma \cdot \tau$;

  Then get n' by reifying the induction hypothesis: $\uparrow_\tau$ `fold$'(C, N, TL)$

  And finally use reflection on the stuck fold to build the element of the model: $\downarrow_\tau$ `fold(c', n', xs_{ne})

□

Theorem 3.1.5 (Evaluation function)**.** *Given a term in* $\Gamma \vdash \sigma$ *and a semantical environment in* $\mathcal{M}\varepsilon(\Delta, \Gamma)$*, one can build a semantical object in* $\mathcal{M}(\Delta, \sigma)$*.*

Proof. Simple induction on the term to be evaluated using the previous lemmas to actually perform the operations in the semantical model. □

Corollary 3.1.6 (Normalization)**.** *For all term t of type $\Gamma \vdash \sigma$, one can compute a normal form t_{nf} in $\Gamma \vdash_{nf} \sigma$. The corresponding function is called* `norm`.

Proof. The evaluation of t in the canonical diagonal environment yields a semantical object T which can then be reified using $\uparrow_\sigma$. □

3.2 Meta-theory

The meta-theory is an ad-hoc extension of the techniques already well explained by Catarina Coquand[9] 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 behave uniformly and to be (extensionally) equal.

Theorem 3.2.1 (Soundness)**.** *For all term t of type $\Gamma \vdash \sigma$, t reduces by $\beta\eta\iota$ to the erasure of* `norm` t.

The soundness proof is here omitted ; it is obviously quite similar to the one for the weak-head normalization procedure while being simpler because of the model storing less information. The proof that this normalization is complete with respect to the equational theory induces by the reduction relation is far more interesting.

Completeness can be summed up by the fact that the interpretation of $\beta\eta\iota$ convertible elements produces semantical objects behaving similarly. This notion of similar behaviour is formalized as *extensional equality* where the domain of the function is limited to *uniform elements* 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.

Setting up logical relations

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 conjunction of the hypotheses – the empty conjunction being $\top$ – is the requirement for the goal to hold. Hence leading to a natural interpretation:

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

Definition The extensional 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 $\texttt{Uni}_\sigma\ T$. They are both mutually defined by induction on the index σ.

Quite unsurprisingly, the unit case is of no interest: all the semantical units are equivalent and uniform.

$$\frac{}{T \equiv_{`1} U} \qquad \frac{}{\mathtt{Uni}_{`1}\ T}$$

Semantical equality for elements with base types is purely syntactical: inhabitants are just bits of data (neutrals) which can be compared using the propositional equality. They are always uniform.

$$\frac{T \equiv U}{T \equiv_{`\mathtt{b}\ k} U} \qquad \frac{}{\mathtt{Uni}_{`\mathtt{b}\ k}\ T}$$

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.

$$\frac{A \equiv_\sigma C \qquad B \equiv_\tau D}{(A,B) \equiv_{\sigma \times \tau} (C,D)} \qquad \frac{\mathtt{Uni}_\sigma\ A \qquad \mathtt{Uni}_\tau\ B}{\mathtt{Uni}_{\sigma \times \tau}\ (A,B)}$$

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).

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

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

In the $`\mathtt{list}\ \sigma$ case, extensional equality is an inductive set basically building the (extensional) diagonal relation on lists of the same type. It is, not suprinsingly, 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.

$$\frac{XS : \mathcal{M}(\Gamma, `\mathtt{list}\ \sigma) \qquad YS : \mathcal{M}(\Gamma, `\mathtt{list}\ \sigma) \qquad EQ_\sigma : \forall \Delta, \mathcal{M}(\Delta,\sigma) \to \mathcal{M}(\Delta,\sigma) \to \mathtt{Set}}{XS \equiv_\sigma^{`\mathtt{list}} YS : \mathtt{Set}}$$

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

$$\frac{\begin{array}{c}xs : xs_1 \equiv xs_2 \qquad YS : YS_1 \equiv_\sigma^{`\mathtt{list}} YS_2 \\ f : \forall (inc : \Delta \supseteq \Gamma)(t : \Delta \vdash_{ne} \tau) \to EQ_\sigma(F_1(inc,t), F_2(inc,t))\end{array}}{\mathtt{mappend}(f,xs,YS) : \mathtt{mappend}(F_1,xs_1,YS_1) \equiv_\sigma^{`\mathtt{list}} \mathtt{mappend}(F_2,xs_2,YS_2)}$$

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.

$$\mathtt{Uni}_{`\mathtt{list}\ \sigma}\ `[] = \top$$

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.

$$\mathtt{Uni}_{\text{`}\mathtt{list}\ \sigma}\ HD\ \text{`}{::}\ TL = \mathtt{Uni}_{\sigma}\ HD \times \mathtt{Uni}_{\text{`}\mathtt{list}\ \sigma}\ TL$$

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.

$$\begin{aligned}&\mathtt{Uni}_{\text{`}\mathtt{list}\ \sigma}\ \mathtt{mappend}_{\tau}(F, xs, YS) = \\ &\quad \forall(inc\colon \Delta \supseteq \Gamma)(t\colon \Delta \vdash_{ne} \tau), \mathtt{Uni}_{\sigma}\ F(inc, t) \\ &\times\ \ \forall inc_1, inc_2, t, wk_{inc_1}\ F(inc_2, t) \equiv_{\sigma} F(inc_2 \cdot inc_1, wk_{inc_1}\ t) \\ &\times\ \ \mathtt{Uni}_{\text{`}\mathtt{list}\ \sigma}\ YS\end{aligned}$$

Remark The reader will notice that $(.\ \equiv_{\sigma}\ .)_{\sigma \in \mathtt{type}_n}$ is a family of equivalence relations. Additionally, she will not have any problem extending these relations in a pointwise fashion to environments as well as checking that all these notions are compatible with weakening.

Proving completeness

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.

Theorem 3.2.2. *If $S \equiv_{\sigma} T$ holds then the reifications of S and T are identical. This theorem is proved mutually with two lemmas:*

- *the reflection of a neutral object is always uniform*
- *weakening and reification commute for uniform objects*

Proof. A simple induction on the type σ of the expressions at hand is enough for this proof to go through. □

Now that we know that all the theorem proving ahead of us will not be meaningless, we can start actually proving 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 3.2.3. *Evaluation in uniform environments produces uniform values. This result is proved mutually with two lemmas:*

Evaluation in semantically equivalent environments produces semantically equivalent values.

Weakening the evaluation of a term is equivalent to evaluating this term in a weakened environment.

Proof. This is proven by induction on the element being evaluated. Introducing intermediate lemmas to deal with the helper functions in a modular manner is a good idea. □

Following the evaluation function's definition, one should prove that combining extensionally equal terms using semantical combinator such as append or map yields extensionally equal terms. This will prove useful when trying to prove equal the evaluation of two related terms whose convertibility proof is using structural rules.

Lemma 3.2.4. *If s and t are two terms in $\Gamma \vdash \sigma$ such that $s \rightsquigarrow_{\beta\eta\iota} t$ and if R is a uniform environment in $\mathcal{M}\varepsilon(\Delta, \Gamma)$ then the evaluation of s in R is extensionally equal to the one of t in R.*

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

Structural rules The case of the structural rule for lambda can be discharged quite simply by an induction hypothesis: indeed a weakened uniform environment is still uniform and the element provided by the extensional equality relation at an arrow type is assumed to be uniform.

The left structural rule for application is trivially discharged by combining the induction hypothesis with lemma 3.2.3 which guarantees that the applied value is indeed uniform. The right structural one works the other way around: the uniformity of the evaluation of the functional part precisely says that application of uniform terms which are extensionally equal (induction hypothesis) yiels semantically equal terms thus proving the goal.

The structure itself of the call graph of $T \equiv_\sigma U$ on product types guarantees that structural rules for pair formers can be discharged by a combination of reflexivity and induction hypothesis while structural rules for projections are taken care of by projecting the appropriate component of the induction hypothesis.

The structural rules for append, map and fold are dealt with by putting together reflexivity proofs and the induction hypothesis using the proofs that these semantical operations yield extensionally equal terms when fed with such kinds of objects.

$\beta\iota$ 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 $\text{`}\lambda\, b$ and its argument x, one starts by proving that the diagonal semantical environment extended with the evaluation of x in R is extensionally equal to the evaluation in R of the diagonal substitution extended with x. 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.

η rules Eta rules definitely are the most complicated ones: except for the ones for product and unit types which can be discharged by reflexivity of the semantical equality, all of them need at least a little bit of theorem proving to go through. The map-id, map-append, append-nil and append-assoc rules can be proven using simple auxiliary lemmas proved by functional induction. □

Theorem 3.2.5 (Completeness)**.** *For all terms t and u of type $\Gamma \vdash \sigma$, if $t \cong_{\beta\eta\iota} u$ then* `norm` $t =$ `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 $t \cong_{\beta\eta\iota} u$, 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 3.2.6 (Decidability of $\cong_{\beta\eta\iota}$)**.** *For all terms t and u of type $\Gamma \vdash \sigma$, $t \cong_{\beta\eta\iota} u$ is decidable.*

Proof. Equality of normal and neutral forms is a simple syntactical problem. Soundness and completeness give means to link (counter)proofs of equality of normal forms to (counter)proofs of $\beta\eta\iota$ equality of the initial terms. □

Remark Proving decidable the propositional equality of normal forms is quite cumbersome in Agda. This could be facilitated by the existence of some kind of *deriving* mecanism for inductive definition. See 5.2 for a modest example of what can be achieved thanks to the internalization of data-type descriptions and the meta-programming it allows.

Showing that the calculus' equational theory is compatible with Agda's one surely is a simple enough sanity check guaranteeing that the chosen definitions somehow make sense[1] but it can also be seen as a way to automatically come up with proofs of equality in Agda; in other words: a tactics.

4.1 Embedding of the calculus in Agda's one

Definition Every finite type in our calculus has an Agda counterpart given that a valuation ρ of its base types is available. We note $[\![\ .\ ]\!]$. this operation of type $\texttt{type}_n \to \texttt{Vec}_n\texttt{Set} \to \texttt{Set}$ defined by induction on its $\texttt{type}_n$ argument:

$$
\begin{array}{lcl}
[\![\ \sigma\ ]\!]\ \rho \Leftarrow & & \texttt{type-induction}(\sigma) \\
[\![\ \text{`}\texttt{b}\ k\ ]\!]\ \rho & = & \rho\ !!\ k \\
[\![\ \text{`}\texttt{1}\ ]\!]\ \rho & = & \top \\
[\![\ \sigma\ \text{`}\times\ \tau\ ]\!]\ \rho & = & [\![\ \sigma\ ]\!]\ \rho \times [\![\ \tau\ ]\!]\ \rho \\
[\![\ \sigma\ \text{`}\to\ \tau\ ]\!]\ \rho & = & [\![\ \sigma\ ]\!]\ \rho \to [\![\ \tau\ ]\!]\ \rho \\
[\![\ \text{`}\texttt{list}\ \sigma\ ]\!]\ \rho & = & \texttt{List}\ ([\![\ \sigma\ ]\!]\ \rho)
\end{array}
$$

This definition is extended to contexts in a pointwise manner thus giving the operation $[\![\ .\ ]\!]\varepsilon$. of type $\texttt{Con}(\texttt{type}_n) \to \texttt{Vec}_n\texttt{Set} \to \texttt{Set}$.

Theorem 4.1.1 (Existence of an embedding)**.** *From any term t of type $\Gamma \vdash \sigma$, one can build an embedding $\lceil\ t\ \rceil$ of t in Agda which produces a $[\![\ \sigma\ ]\!]\ \rho$ when fed with a $[\![\ \Gamma\ ]\!]\varepsilon\ \rho$.*

Proof. Straightforward by induction on the structure of t. □

This embedding is trivially extended to parallel substitutions ts of type $\Delta \vdash\varepsilon\ \Gamma$ in a pointwise fashion. We note $\lceil\ .\ \rceil\varepsilon$ this translation.

Theorem 4.1.2 (Soundness of the reduction relation)**.** *In* Agda + Functional Extensionality[2] *terms of type $\Gamma \vdash \sigma$ convertible in the source language, when embedded in Agda and fed with the same $[\![\ \Gamma\ ]\!]\varepsilon\ \rho$, evaluate to propositionally equal objects.*

Proof. Let s and t be two terms of type $\Gamma \vdash \sigma$ such that $s \rightsquigarrow_{\beta\eta\iota} t$, let R be a realiser of the environmentna $[\![\ \Gamma\ ]\!]\varepsilon\ \rho$. The proof that $\lceil\ s\ \rceil(R) \equiv \lceil\ t\ \rceil(R)$ is by induction on the shape of the proof that $s \rightsquigarrow_{\beta\eta\iota} t$.

The structural rules are dealt with quite easily combining the induction hypotheses with the appropriate congruence rules. The only tricky case is the structural rule allowing reductions under $\text{`}\lambda$s where functional extensionality is needed.

[1] Indeed if a non-sensical proof can be built in our calculus then this construction guarantees that it translates down to one in Agda which would be most regrettable.

[2] We are forced to use extensionality because we are identifying under binders terms that are only equal propositionally e.g. we consider that $\lambda xs \to \texttt{map}(\texttt{id}_A, xs) \equiv \texttt{id}_{\texttt{List}A}$ which is not provable in core Agda.

The ι rules are similarly non-problematic: the embedding of a ι redex and its reduct are exactly equal thus allowing the case to be discharged by invoking the propositional equality's reflexivity.

On the other hand β reduction is not trivially correct: one has to prove a more general commutation lemma relating the application of a substitution to a term and its evaluation in an environment basically consisting of the embedding of the substitution: $\lceil\ t[\rho]\ \rceil(R) \equiv \lceil\ t\ \rceil(\lceil\ \rho\ \rceil\varepsilon(R))$.

Only two of the standardization rules (also called η rules) are discharged by invoking the reflexivity of the propositional equality (namely the ones for types $\text{`}\mathtt{1}$ and $\_\ \text{`}\times\ \_$). Almost all the ones mentioning lists necessitate nothing more than a simple induction following the reduction behaviour of the functions involved (be it either map, fold or append). The only three remaining cases are map fusion, fold-map fusion and η-expansion of functions.

η-expansion of functions is dealt with by using functional extensionality in order to apply a more general lemma: the translation of a weakened term in an environment R amounts to the translation of the original term in an environment R' equal to R purged of all the realizers of types added to the context by the weakening.

In the case of (fold-)map fusions, a simple induction can prove the result on semantical objects, the only missing step being a commutation lemma between composition at the term level and composition after embedding. It is proven by combining with functional extensionality two instances of the lemma described in the η-expansion case. □

Corollary 4.1.3 (Existence of a tactics)**.** *For all terms lhs and rhs of type* $\Gamma \vdash \sigma$ *and for all valuation* ρ *of their base types and* R *of their free variables, the following proposition can be discharged automatically whenever it holds:*

$$\lceil\ lhs\ \rceil(R) \equiv \lceil\ rhs\ \rceil(R)$$

Proof. A trivial structural induction on two terms' convertibility derivation using the previous theorem 4.1.2 provides us with a proof that whenever $s \cong_{\beta\eta\iota} t$ holds then so does $\lceil\ s\ \rceil(R) \equiv \lceil\ t\ \rceil(R)$.

The convertibility test described in corollary 3.2.6 allows us to decide if $s \cong_{\beta\eta\iota} t$ is provable. Depending whether it holds or not, one can ask the user to provide either a (trivial) proof of $\top$ or a proof of $\bot$ hence always being able to discharge the goal.

This tactics is called `solve` and is invoked with s, t, ρ, R and `tt` thus letting Agda either typecheck the term and discharge the goal whenever it is provable or complain that `tt` is not of type $\bot$ otherwise. □

Example Let `swap` be the function transforming pairs of type $A \times B$ into pairs of type $B \times A$. It can be represented in our calculus by the term $\text{`}\mathtt{swap} = \text{`}\lambda(\text{`}\pi_2\ \text{`}\mathtt{v}\,0\ \text{`},\ \text{`}\pi_1\ \text{`}\mathtt{v}\,0)$ of principal type $\Gamma \vdash \sigma\ \text{`}\times\ \tau\ \text{`}\to\ \tau\ \text{`}\times\ \sigma$ for any Γ, σ and τ.

Then for any A and B: `Set` and any list xs of $A \times B$ pairs, one can prove that $\mathtt{map}(\mathtt{swap}, \mathtt{map}(\mathtt{swap}, xs)) \equiv xs$ holds true by simply invoking the solver:

$$\mathtt{solve}(\text{`}\mathtt{map}(\text{`}\mathtt{swap}, \text{`}\mathtt{map}(\text{`}\mathtt{swap}, \text{`}\mathtt{v}\,0)), \text{`}\mathtt{v}\,0, \underbrace{[A, B]}_{\rho}, \underbrace{(\mathtt{tt}, xs)}_{R}, \mathtt{tt})$$

Looking for new models with appropriate reduction behaviours is a non-trivial problem and when designing a candidate hopefully producing a simple and bug-free formalization, one is bound to explore a throng of alternatives which end up being discarded.

The few ones presented here are odd birds taken from this bunch. They were not cultivated enough to become full-blown developments like the main subject of this presentation but the concepts they are based on are interesting enough to deserve being articulated if only once.

5.1 Baking is fun

The Kripke-style semantics $\mathcal{M}$ is structured very similarly to the way the logical relation implying soundness is built. It is therefore possible to bake the logical relation's invariant in the model itself thus guaranteeing the evaluation function to be written sound *by construction*. When interacting with Agda, this provides the developper with a more secure working environment: instead of implementing functions ultimately proven correct, his effort is driven by the typechecker giving instantaneous feedback on his code. And once the algorithm is implemented, there is no need for a mathematician's analysis proving the algorithm sound *a posteriori*.

The modified model is obtained by enriching the semantical objects with well-typed terms (called decorative terms[1]) which are, intuitively, terms the semantical objects are reducts of. As a consequence, this construction has a Kripke flavor with respect to two parameters: it is upward closed under context extension but also under $\beta\iota$-expansion and η-contraction of decorations.

Definition The definition of this decorated model highlights the differences with the usual one by putting the refinements in red boxes. The model is indexed by a context Γ, a type σ and a well-typed term $t : \Gamma \vdash \sigma$. It is defined by induction on the type:

$$
\begin{array}{lcl}
\mathcal{M}(\Gamma,\ \sigma\ ,\ t\ ) & \Leftarrow & \texttt{type-induction}(\sigma) \\
\mathcal{M}(\Gamma,\ \text{`}\texttt{1}\ ,\ \_\ ) & = & \top \\
\mathcal{M}(\Gamma,\ \text{`}\texttt{b}\ k\ ,\ t\ ) & = & (t_{ne} : \Gamma \vdash_{ne} \text{`}\texttt{b}\ k) \times t \rightsquigarrow^*_{\beta\eta\iota} t_{ne} \\
\mathcal{M}(\Gamma,\ \sigma\ \text{`}\times\ \tau\ ,\ p\ ) & = & (a : \Gamma \vdash \sigma) \times (b : \Gamma \vdash \tau) \times p \rightsquigarrow^*_{\beta\eta\iota} (a, b) \times \\
 & & \mathcal{M}(\Gamma, \sigma,\ a\ ) \times \mathcal{M}(\Gamma, \tau,\ b\ ) \\
\mathcal{M}(\Gamma,\ \sigma\ \text{`}\rightarrow\ \tau\ ,\ f\ ) & = & \forall \Delta \supseteq \Gamma,\ \forall (x : \Delta \vdash \sigma)\ ,\ \forall y \rightsquigarrow^*_{\beta\eta\iota} f\ \text{`}\$\ x\ \rightarrow \\
 & & \mathcal{M}(\Delta, \sigma,\ x\ ) \rightarrow \mathcal{M}(\Delta, \tau,\ y\ ) \\
\mathcal{M}(\Gamma,\ \text{`}\texttt{list}\ \sigma\ ,\ xs\ ) & = & \mathcal{L}(\Gamma, \sigma, \Vdash_\sigma,\ xs\ )
\end{array}
$$

where $\mathcal{L}(\Gamma, \sigma, \Vdash_\sigma,\ xs\ )$ is and inductive definition parametric in its arguments which definition is based on the one of the simpler model:

[1] It is not an accident that their name is similar to McBride's ornaments[18] given that both of them are refinement of an existing structure. The formal connection is however yet to be made formal.

$$\frac{\Vdash_\sigma : (\Delta : \mathtt{Con}(\mathtt{type}_n)) \to \Delta \vdash \sigma \to \mathtt{Set} \qquad xs : \Gamma \vdash \text{`}\mathtt{list}\ \sigma}{\mathcal{L}(\Gamma, \sigma, \Vdash_\sigma, xs) : \mathtt{Set}}$$

$$\frac{xs \rightsquigarrow^*_{\beta\eta\iota} \text{`}[]}{\text{`}[] : \mathcal{L}(\Gamma, \sigma, \Vdash_\sigma, xs)} \qquad \frac{xs \rightsquigarrow^*_{\beta\eta\iota} hd\ \text{`}::\ tl \qquad HD : \Vdash_\sigma (\Gamma, hd) \qquad TL : \mathcal{L}(\Gamma, \sigma, \Vdash_\sigma, tl)]}{HD\ \text{`}::\ TL : \mathcal{L}(\Gamma, \sigma, \Vdash_\sigma, xs)}$$

$$\frac{xs \rightsquigarrow^*_{\beta\eta\iota} \text{`}\mathtt{map}(f, ts)\ \text{`}{+}{+}\ ys \qquad ts : \Gamma \vdash_{ne} \text{`}\mathtt{list}\ \tau \qquad YS : \mathcal{L}(\Gamma, \sigma, \Vdash_\sigma, ys) \qquad F : \forall \Delta \supseteq \Gamma, (t : \Delta \vdash_{ne} \tau) \to \Vdash_\sigma (\Delta, f\ \text{`}\$\ t)}{\mathtt{mappend}(f, F, ts, YS) : \mathcal{L}(\Gamma, \sigma, \Vdash_\sigma, xs)}$$

Proposition 5.1.1 (Closure)**.** *The model is closed under expansion of the decorative term with respect to the reduction relation.*

Proof. By a simple induction on the type followed by a case analysis on the term in the list case. □

Because the model now internalizes some of the proofs, it is not possible to write the pair reify / reflect on its own anymore. Let us recall the $\sigma\ \text{`}{\to}\ \tau$ case for $\downarrow_\sigma$ in the non decorated case:

$$\downarrow_{\sigma \hookrightarrow \tau} f = \lambda\Delta\ inc\ x.{\downarrow_\tau}(wk_{inc}(f)\ \text{`}\$\ {\uparrow_\sigma}x)$$

In the decorated case, one has to take care of an extra y which reduces to the application of f to the decoration of x. It calls for the use of the closure lemma 5.1.1 to cast the recursively built application of f to the reification of x. But at this point in time there's no information available on the relationship between x's decoration and the value obtained by reifying x.

The solution is to not only build two functions by mutual induction but rather three of them: reflection, reification and the proof that reification yields a reduct of the decoration.

Lemma 5.1.2 (Coherent reify and reflect)**.** *Mutually defined processes allow normal forms $x_{nf} : \Gamma \vdash_{nf} \sigma$ to be extracted from elements of the model $\mathcal{M}(\Gamma, \sigma, x)$ such that $x \rightsquigarrow^*_{\beta\eta\iota} x_{nf}$ while neutral forms $x_{ne} : \Gamma \vdash_{ne} \sigma$ can be turned into elements of the model $\mathcal{M}(\Gamma, \sigma, x_{ne})$.*

Proof. The construction is done by induction on the type σ in a fashion very similar to the versions working on the non decorated models. □

Limitations

One does get soundness for free by internalizing the logical relation but all of this has a cost: extra casts are inserted all over the place at evaluation time! This makes the equational reasoning involved in proving completeness far more tedious ; indeed before actually reaching the interesting bits of the values mentionned, one has to deal with all these coercions first.

5.2 A language for describing datatypes

Using a universe of description for indexed datatypes, it is possible to design some kind of *deriving* mecanism simplifying the proofs that equality is decidable for the terms of a given datatype. Noticing that the decidability of $\cong_{\beta\eta\iota}$ for our calculus relies on the decidability of the propositional equality for normal and neutral forms, it gives a quite good opportunity to test the usability of such a system.

Here is the grammar used to define the class of strictly positive functors used in this example as well as the interpretation function building the corresponding I to $\mathtt{Set}$ functor.

$$\frac{I : \mathtt{Set}}{\mathtt{Desc}_I : \mathtt{Set}_1} \qquad [\![\ .\ ]\!] : \mathtt{Desc}_I \to (I \to \mathtt{Set}) \to I \to \mathtt{Set}$$

$$\frac{i : I \qquad d : \mathtt{Desc}_I}{\mathtt{rec}(i,d) : \mathtt{Desc}_I} \qquad [\![\mathtt{rec}(i,d)]\!](R,j) = R(i) \times [\![d]\!](R,j)$$

$$\frac{i : I}{\mathtt{ret}(i) : \mathtt{Desc}_I} \qquad [\![\mathtt{ret}(i)]\!](R,j) = i \equiv j$$

$$\frac{d_1 : \mathtt{Desc}_I \qquad d_2 : \mathtt{Desc}_I}{d_1 \,`\uplus\, d_2 : \mathtt{Desc}_I} \qquad [\![d_1 \,`\uplus\, d_2]\!](R,j) = [\![d_1]\!](R,j) \uplus [\![d_2]\!](R,j)$$

$$\frac{A : \mathtt{Set} \qquad d_A : (a : A) \to \mathtt{Desc}_I}{`\Sigma(A, d_A) : \mathtt{Desc}_I} \qquad [\![`\Sigma(A, d_A)]\!](R,j) = (a : A) \times [\![d_A(a)]\!](R,j)$$

A datatype $`\mu\ d$ with only one constructor then gives the mean to take the fixpoint of such a functor:

$$\frac{d : \mathtt{Desc}_I \qquad i : I}{`\mu\ \langle d, i\rangle : \mathtt{Set}} \qquad \frac{t : [\![d]\!](\lambda i.`\mu\ \langle d, i\rangle, i)}{\mathtt{in}(t) : `\mu\ \langle d, i\rangle}$$

Reconstructing the inductive datatypes used

To keep this example short and readable, we are going to focus on a simply-typed calculus with a base type. Dagand proved in his paper on elaboration of datatypes declaration to codes[11] that this description-based approach is versatile enough to allow more complex systems to be encoded.

Contexts $Con(A)$ of elements in A are either empty or the combination of a head $(a : A)$ and a tail $Con(A)$. Their are not indexed thus the use of unit as the set of indices:

$$Con(A) = `\mu\ \langle `\uplus \quad `\Sigma(A, \lambda\_.\, \mathtt{rec}(\mathtt{tt}, \begin{matrix}\mathtt{ret\,tt}\\ \mathtt{ret\,tt}\end{matrix})) \quad , \mathtt{tt}\rangle$$

Using Andjelkovic and Gundry's patterns[5], we can recover a syntax mimicking the presence of constructors for these different datatypes: unlike other definitions, patterns can be used on the left hand side of an equation. ε will stand for $\mathtt{con}(\iota_1 refl)$ while $\Gamma \cdot \sigma$ will mean $\mathtt{con}(\iota_2(\sigma, \Gamma, refl))$.

The membership predicate is indexed by a context $\Gamma : Con(A)$ and is parametric in A and an element $a : A$. It is defined as either *here* if a is the head of Γ or *there* if a is further down the context. The subscript A will be omitted whenever possible.

$$a \in_A \Gamma = \text{`}\mu \left\langle \text{`}\uplus \begin{array}{ll} \text{`}\Sigma(Con(A), \lambda\Gamma. & \mathtt{ret}(\Gamma \cdot a) \\ \text{`}\Sigma(Con(A) \times A, \lambda(\Gamma, b).\, \mathtt{rec}(\Gamma, & \mathtt{ret}(\Gamma \cdot b))) \end{array} , \Gamma \right\rangle$$

A type `type` is either a base type or an arrow type composed of two types: the domain and the codomain of the arrow. There is no need for indexing here either.

$$\mathtt{type} = \text{`}\mu \left\langle \text{`}\uplus \begin{array}{ll} & \mathtt{ret\,tt} \\ \mathtt{rec}(\mathtt{tt}, \mathtt{rec}(\mathtt{tt}, & \mathtt{ret\,tt})) \end{array} , \mathtt{tt} \right\rangle$$

A term $\Gamma \vdash \sigma$ is indexed by a context Γ and a type σ. It can either be:

A variable i.e. a context, a type and a proof that this type belongs to that context:

$var = \text{`}\Sigma(\mathtt{Con}(\mathtt{type}) \times \mathtt{type}, \lambda(\Gamma, \sigma).\, \text{`}\Sigma(\sigma \in \Gamma, \lambda\_.\, \mathtt{ret}(\Gamma, \sigma)))$

A lambda abstraction i.e. a context, two types and a body of the right type given the extended context:

$lam = \text{`}\Sigma(\mathtt{Con}(\mathtt{type}) \times \mathtt{type} \times \mathtt{type}, \lambda(\Gamma, \sigma, \tau).\, \mathtt{rec}((\Gamma \cdot \sigma, \tau), \mathtt{ret}(\Gamma, \sigma \text{`}\to \tau)))$

The application of a function to an argument i.e. a context, two types, a functional and and argument whose type matches the functional's domain:

$app = \text{`}\Sigma(\mathtt{Con}(\mathtt{type}) \times \mathtt{type} \times \mathtt{type}, \lambda(\Gamma, \sigma, \tau).\, \mathtt{rec}((\Gamma, \sigma \text{`}\to \tau), \mathtt{rec}((\Gamma, \sigma), \mathtt{ret}(\Gamma, \tau))))$

Hence the following description for terms:

$$\Gamma \vdash \sigma = \text{`}\mu \left\langle \text{`}\biguplus \begin{array}{l} var \\ lam \\ app \end{array} , (\Gamma, \sigma) \right\rangle$$

The description of neutral and normal forms is quite similar to the one of terms except for one subtlety: these definitions are mutual. This is solved by, first, defining a set with two distinct elements (`ne` and `nf`, called tags) and then use these tags as indices restricting the shape an expression can have.

The definition of the normalization by evaluation is then absolutely identical to the one where datatypes are declared in a more classical way. The only difference being that decidability of syntactical equality comes more or less for free for all the structure used.

Limitations

There are no real limitations here except the fact that, unlike in Epigram[8], these concepts are not internalized in Agda. As a consequence, the way the goals of an unfinished proof are displayed is quite appalling which makes it painful to work on.

5.3 A calculus with a universe of datatypes

As demonstrated in the previous section, it is possible to describe the notion of inductive definition in a calculus by using codes for polynomial functors and interpreting them as such. This extension of the calculus with datatypes is compatible with building a normalization by evaluation procedure. In this section we limit ourselves to skeletons i.e. datatypes with no parameters or indices but the reader should be able to extend all the results to (at least)

datatypes equipped with parameters. During this formalization she may find Agda's complaints about termination painfully recurrent ; inlining a couple of definitions usually solves the problem.

cite supercompilation as a proof of termination

In addition to the machinery used to represent the simply-typed lambda calculus, one introduces descriptions as a new inductive definition and functor applications and fixpoints as new type formers.

$$d \,:\, \texttt{Desc} ::= \text{`}\texttt{1} \mid d_1 \text{`}\!+ d_2 \mid d_1 \text{`}\!\times d_2 \mid \text{`}\texttt{X}$$

$$\frac{d\colon \texttt{Desc}}{\text{`}\mu\ d\colon \texttt{type}_n} \qquad \frac{d\colon \texttt{Desc} \qquad \sigma\colon \texttt{type}_n}{\texttt{F[}\ d\ \texttt{]}\ \sigma\colon \texttt{type}_n}$$

Then one has to add new constructors for terms allowing to build inhabitants of a functor or eliminate them. The sum of two functors is here used as an example of such rules:

$$\frac{p_1\colon \Gamma \vdash \texttt{F[}\ d_1\ \texttt{]}\ \sigma}{\text{`}\iota_1(d_2, p_1)\colon \Gamma \vdash \texttt{F[}\ d_1 \text{`}\!+ d_2\ \texttt{]}\ \sigma} \qquad \frac{p_2\colon \Gamma \vdash \texttt{F[}\ d_2\ \texttt{]}\ \sigma}{\text{`}\iota_2(d_1, p_2)\colon \Gamma \vdash \texttt{F[}\ d_1 \text{`}\!+ d_2\ \texttt{]}\ \sigma}$$

$$\frac{f_1\colon \Gamma \vdash \texttt{F[}\ d_1\ \texttt{]}\ \sigma \text{`}\!\to \tau \qquad f_2\colon \Gamma \vdash \texttt{F[}\ d_2\ \texttt{]}\ \sigma \text{`}\!\to \tau}{\text{`}\texttt{+-elim}(f_1, f_2)\colon \Gamma \vdash \texttt{F[}\ d_1 \text{`}\!+ d_2\ \texttt{]}\ \sigma \text{`}\!\to \tau}$$

Finally the addition of a fold operator and a map lifting functions on functors [2] gives this small language an expressive enough core.

$$\frac{f\colon \Gamma \vdash \sigma \text{`}\!\to \tau \qquad x\colon \Gamma \vdash \texttt{F[}\ d\ \texttt{]}\ \sigma}{\text{`}\texttt{map}(f, x)\colon \Gamma \vdash \texttt{F[}\ d\ \texttt{]}\ \tau}$$

$$\frac{alg\colon \Gamma \vdash \texttt{F[}\ d\ \texttt{]}\ \sigma \text{`}\!\to \sigma \qquad x\colon \Gamma \vdash \text{`}\mu\ d}{\text{`}\texttt{fold}(alg, x)\colon \Gamma \vdash \sigma}$$

The model definition is then quite similar to the one for our main calculus: it is defined by induction on the type index and functors applied to a type are dealt with in an inductive definition parametric in its arguments. The only way to lift a neutral inhabitant of a functor being to either use η-expansion (e.g. in the case of a product) or a stuck map.

[2] Notice that when adding parameters to the datatypes, one can define another map function: the one acting on the parameters rather than on the variable of the polynomial functor!

6.1 Relax the structural constraints of the model

The current model for full-normalization as well as the definition of normal forms exhibit artifacts characteristic of the arbitrary choice that was made when picking a representative of the equivalence classes (e.g. right-nesting of appends ending with and empty list, map-id expansions on neutrals). It would be nicer to have relaxed notions of normal forms and semantical lists and simplify away these right units, or identity mapping rather than adding them.

6.2 Quoting machinery for the tactics

In the current implementation of the solver (see chapter 4 – A tactics for Agda), the reification of the goal one is trying to prove has to be performed by hand which can be quite tedious and is of no particular interest to the programmer. This work should be assigned to the machine rather than the user.

In his Msc. thesis on reflection in Agda, Paul van der Walt describes a reification algorithm for a simply-typed lambda calculus ([21], Chapter 5) which could probably be adapted to our use case. He proceeds in two consecutive steps: the first one building raw terms annotated with type information in crucial places (i.e. at binding time) using the quotation mechanism and the second one extracting a well-typed term from a raw one. It currently necessitates patching Agda's reification mechanism to insert these extra pieces of information.

6.3 Extending the calculus

Be it either in terms of supported equational theories, or in terms of complexity of the type system, this experiment should be pushed further. Partial solutions for both of these problems already exsits:

Atkey's Foveran implements monad laws for indexed descriptions[20].

Normalization by evaluation already has been extended in some ways to Martin-Löf type theories by Abel and al.[2, 3] but these presentations say nothing about data let alone richer equational theories.

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

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

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

[4] Andreas Abel, Thierry Coquand, and Peter Dybjer. Verifying a semantic $\beta\eta$-conversion test for martin-löf type theory mathematics of program construction. In Philippe Audebaud and Christine Paulin-Mohring, editors, *Mathematics of Program Construction*, volume 5133 of *Lecture Notes in Computer Science*, chapter 4, pages 29–56. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2008.

[5] Stevan Andjelkovic and Adam Gundry. Support for pattern synonyms in Agda. Agda bugtracker - Feature request #495, March 2012.

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

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

[8] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. *SIGPLAN Not.*, 45(9):3–14, September 2010.

[9] Catarina 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.

[10] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical. Structures in Comp. Sci.*, 7:75–94, February 1997.

[11] Pierre-Evariste Dagand and Conor McBride. Elaborating Inductive Definitions. *ArXiv e-prints*, October 2012.

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

[13] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Mogensen, and Peter Thiemann, editors, *Partial Evaluation*, volume 1706 of *Lecture Notes in Computer Science*, chapter 16, pages 367–411. Springer Berlin / Heidelberg, Berlin, Heidelberg, 1999.

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

[15] Ralf Hinze, Thomas Harper, and DanielW James. Theory and practice of fusion. In Jurriaan Hage and MarcoT Morazán, editors, *Implementation and Application of Functional Languages*, volume 6647 of *Lecture Notes in Computer Science*, pages 19–37. Springer Berlin Heidelberg, 2011.

[16] Chantal Keller and Thorsten 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.

[17] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, Naples, 1984.

[18] Conor McBride. Ornamental algebras, algebraic ornaments. 2010.

[19] Ulf Norell. Dependently typed programming in Agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008.

[20] Atkey Robert. A type checker that knows its monad from its elbow. Blog post, December 2011.

[21] Paul van der Walt. Reflection in Agda. Master's thesis, Universiteit Utrecht, 2012.