Staged computations in types

Mathieu Boespflug

McGill University
mboes@cs.mcgill.ca

Abstract. Two-level formal systems segregate a specialized language for convenient higher order representations of object logics while still allowing for computation and powerful reasoning principles to support formal metatheory. But such a strict segregation between two entirely distinct levels limits potential for reuse of declarations between levels. More importantly, it can moreover be useful to reason about computations, not just representations, and to compute large terms and long derivations from the representation layer rather than writing them out in full. We show how to extend a two level system, allowing for computations in propositions, but without compromising the adequacy of encodings into the representation layer. The enabling ingredient is to distinguish between values and computations, in the style of [1], and to only allow embedding values into the representation level. We demonstrate through several examples how our extended system offers an excellent framework for proofs by reflection. This style of proof lets a user define ad hoc, domain specific decision procedures safely, without increasing the size of the trusted base. We show type safety of our system, and reduce consistency to that of a simpler system without permutation rules.

1 Introduction

1.1 A shallow embedding is deep one level up

Formal metatheory is the cornerstone of sound programming languages and proof systems research — a proposal for a new language of proofs, formulas or programs only takes credence once a reasonable argument can be made that this new language behaves well and realizes all the right properties. Reasoning about a new language in a formal system, which we'll qualify as the *logical framework*, presupposes that there exists means to express this language in the logical framework. For any non-trivial language, the typical choices are to formulate a *deep embedding* of the object language within the language of the logical framework (or *metalanguage*), encoding expressions of the object language as data in the metalanguage, or a *shallow embedding*, where expressions and propositions of the object language are mapped directly to meta level expressions and propositions. Shallow embeddings, for all their convenience (inherited substitution principles, small size of encoded terms), are not suitable for all types of metatheoretic reasoning: one cannot, for instance, readily compute the size of an expression, and

recovering good induction principles that a deep embedding provides can be tricky.

In the BELUGA system [2], we get the best of both worlds, because this particular logical framework discriminates between two disinct languages, living in two different levels: the LF language, which is the language of the *data level* in which we formulate shallow embeddings, on top of which we are offered a language of computations that allows us to express proofs. We call *proof level* the level at which these computations live. The latter level is the meta level of the former, in that terms and propositions of the data level can be manipulated as data in the proof level. For example, the following signature forms a shallow embedding of higher order logic (HOL) and a fragment of its deductive system in LF[1]:

```
o : type.   ι : type.      eq : ι → ι → o.
                           eq/beta : |- (eq (app (lam (λx. M x)) N) (M N)).
lam : (ι → ι) → ι.         eq/trans : |- (eq M M') → |- (eq M' M'')
app : ι → ι → ι.                                   → |- (eq M M'').

|- : o → type.
```

HOL has only two base types: the type o of propositions and the type ι of HOL terms. At the proof level, we can write a proof that the open term $(\lambda x.x\ x)\ (\lambda z.x\ y)$ is equal to $y\ y$ under HOL's notion of equality. We do this not in the host language LF, but in the language of computations (which we will define more formally in Section 3):

```
let prop1 : [y : ι. eq (app (λx. x x) (λz. app z y)) (app y y)] =
    [. eq/trans eq/beta (eq/trans eq/beta eq/beta)];
```

The square brackets denote boxes; they serve to lift LF types (resp. terms) to proof level types (resp expressions). The beginning of a box always lists explicitly the free variables (and their types) of the object in the box. Notice that at the proof level, we can safely manipulate any term at the data level, including open terms, just as we would have been able to do in LF alone had we gone for a deep embedding. The advantage of the two-level approach is that a shallow embedding at the data level is a deep embedding at the proof level, so at the proof level we may do everything that a deep embedding affords us, while also retaining the free substitution principles that shallow embeddings and other instances of higher order abstract syntax (HOAS) provide.

1.2 A tale of two function spaces

To wit, the following code demonstrates how we can define in this system a function that computes the size of an HOL term shallowly embedded in LF:

```
nat : type = (ι → ι) → ι → ι.     % A type synonym.
zero : nat.
succ : nat → nat.

schema ctx = ι;
```

[1] See [3] for a nice and short overview of HOL.

```
rec size : (g : ctx) [g. ι] → [. nat] =
  fn m ⇒case m of
    | [g. #p..] ⇒ [. succ zero]
    | [g. lam (λx. M..x)] ⇒
      let [. N] = size [g,x : ι. M..x] in [. succ N]
    | [g. app (M1..) (M2..)] ⇒
      let [. N1] = size [g. M1..] in
      let [. N2] = size [g. M2..] in
      let [. N3] = plus [. N1] [.N2] in [. succ N3];
```

In HOL, natural numbers are usually represented as Church numerals, that is to say as higher order (data level) functions. The (proof level) function `size`[2] proceeds by case analysis on an LF object — indeed these objects are data (at the proof level). They need not be closed, and in general live in a context `g`, whose schema `ctx` says that all free variables of an LF object living in `g` are of type ι. The *parameter variable* `#p` in the pattern of the first clause matches any LF variable. *Metavariables* appearing in patterns are always capitalized and match any term whose set of free variables fits within the boundaries set forth by the substitution suffixed to each of them: if it is the identity substitution (denoted `..`) any term with free variables in the context `g` will be matched, likewise for any extension of the identity substitution (such as `..x`) and some extension of `g` (such as `[g,x : ι]`).

But now that we have a function to compute the size of a term, it is natural to ask whether we can reason formally about the properties of this function. For example, we would like to be able to prove that the size function commutes with the `lam` constructor. But since our object theory HOL already formalizes what it means to be a number and what it means for two things to be equal, we would like to reuse these in stating this lemma. More precisely, we would like to prove the following lemma about `size`:

```
size_lam : (g : ctx) let [. N] = size [g. lam (λx. M..x)] in
                     let [. N'] = size [g,x : ι. M..x] in
                     [. eq N (succ N')];
```

That is, we would like to be able to state properties not just about data level entities, but also about the results of computations on these data level entities, all the while without duplicating any theories that might be preexisting in the data level. Moreover (as we shall see in Section 5), adding this feature to a two-level system has the pleasant side effect of letting us replace potentially very large data level objects with computations at the proof level instead. We propose in Section 3 a general and well behaved mechanism that achieves this goal. The idea is to allow demoting proof level *values* down to the data level, but not proof level *computations*. We structure our calculus in a monadic style to distinguish values from computations, much in the same way as Moggi's computational λ-calculus [1].

[2] One can tell the level of a declaration by its terminator token: a "." (resp. ";") marks the end of a data level (resp. proof level) declaration.

1.3 On the relevance of two-level systems

Before delving much deeper into the technical details, one might wonder what the fuss over two-level systems is. We are, after all, seeking to blur in a controlled fashion the strict separation between levels in these systems, so one might ask why start with two levels rather than just one to begin with? A key issue is the adequacy of embeddings, meaning that elements of the object language are in a compositional bijection with terms in the metalanguage of the corresponding type. If we have but one function space, the function space of proofs, and reuse this one function space to encode object languages in a higher order fashion, then it becomes difficult to rule out the existence of so-called *exotic terms* in the encoding, since with the help of some syntactic constructs necessary for a proof language, such as case analyses, it becomes possible to construct terms of the appropriate type that do not correspond to anything in the object language. One salient feature of this work is that it does not compromise encodings' adequacy, because LF terms are still canonical, as is the case in Canonical LF [4].

1.4 Outline

We start with a brief characterization of the data level in Section 2. The features we propose in this system are generic in the data level language, so we detail the requirements that we impose of the data layer, without committing to a particular language. A type system is given in Section 3, about which we show a number of properties (Section 4), culminating in type safety. With the core theory laid out, we discuss some use cases of our two-level system, focusing in particular on proofs by reflection (Section 5).

2 The data level, abstractly

LF makes for a fine data level language, but in this section we will ask only of the data level language that it provide us with a suitable notion of terms, types, contexts and substitutions. We assume that we can meaningfully lift these entities into the meta level, at which point we call them *meta* entities (*e.g.* a meta type) to distinguish them from the similar concepts of the meta language. We use C to refer to meta terms, U for meta types, and X to range over all of meta variables, parameter variables and context variables. In the course of analyzing a meta term, we may learn something about the types of the meta variables, and moreover we will want to relate meta types in the form of equality assumptions. A meta context is a package of such information:

$$\text{Meta subst. } \theta ::= \cdot \mid \theta, C/X \qquad \text{Meta contexts } \Delta ::= \cdot \mid \Delta, X{:}U \mid \Delta, U_1 = U_2$$

Meta variables and parameter variables always occur with an associated substitution, as we have seen in Section 1. We write $\mathsf{id}(X)$ for an occurrence of X associated to the identity substitution. We write $[\![\theta]\!]\cdot$ for the application of the (simualtaneous) meta susbtitution to any entity. Some of these substitutions arise as the most general unifier of type meta types, a fact that we will write

as $\Delta \vdash U_1 \doteq U_2/(\Delta, \theta)$, meaning θ unifies types U_1, U_2 living in context Δ and takes them to context Δ'.

Finally, we write $\Delta \vdash C : U$ for the judgement expressing that C is of type U, and assume the rules that justify it as given. The domains of meta substitutions is normally determined by a pure meta context of typing assumptions, so we define the following additional substitution well formation rule to handle equality assumptions:

$$\frac{\Delta' \vdash \theta : \Delta}{\Delta', [\![\theta]\!]U_1 = [\![\theta]\!]U_2 \vdash \theta : \Delta, U_1 = U_2}$$

3 A proof language for contextual objects

We define a language of computations, separate from the data level's language, whose syntax is defined formally below:

Types	$T ::= U \mid T_1 \to T_2 \mid \Pi X{:}U.T \mid \mathsf{let}\ X : U = E\ \mathsf{in}\ T$
Expressions	$E ::= y \mid C \mid E_1\, E_2 \mid \mathsf{fn}\ y.\, E \mid \Lambda X.\, E \mid \mathsf{rec}\ f.E \mid \mathsf{case}\ E\ \mathsf{of}\ \vec{B}$
Branches	$B ::= \Delta\ .\ C \mapsto E$
Contexts	$\Gamma ::= \cdot \mid \Gamma, y{:}T \mid \Gamma, E \rightsquigarrow C : U$

This language is a domain-free dependently typed λ-calculus, with the addition of a fixpoint construction for writing recursive functions and a case analysis construct on terms of base type, *i.e.* data level values. Note that we understand $U \to T$ and $\Pi X{:}U.T$ as two completely different function spaces: the first is the type of programs, of the form $\mathsf{fn}\ y.\, E$, while the latter is the type of abstractions over a meta type index, written $\Lambda X.\, E$. In other words, we only have data level dependencies in proof level types.

The language given here is largely identical to previous presentations [5–7], but for the addition of the "let-in" construct in the sublanguage of types. This construct is to be understood as the application of a computation E to the *Kleisli extension* of a function from values to computations [1], or more colloquially as the "bind" of a monad. Indeed, the reduction rules for this construct, given in Figure 1, is directly justified by the equational theory of monads.

The other extension is that we allow storing equality assumptions in the expression context (just as we have in the meta context). These equality assumptions arise from a case analysis on a scrutinee E_s. In each branch, we can exploit statically, during typing, what we learn about the dynamic behaviour of E when selecting a branch $\vec{B}_i$: that the value of E at runtime must necessarily match C_i. Therefore, on this assumption, we can rewrite any occurrence of E_s in types and replace it with C_i.

3.1 Term and type equivalence

Since with the addition of the "let-in" construct, computational expressions may now appear in types, type equivalence now needs to be defined modulo expression equivalence. Two expressions are equivalent if they compute to the same normal

$\boxed{T \longrightarrow T'}$ Type T reduces to T', the contextual closure of the following relation

$$\begin{array}{rl}
\mathsf{let}\ X : U = E\ \mathsf{in}\ T & \longrightarrow \mathsf{let}\ X : U = E'\ \mathsf{in}\ T \qquad \text{if } E \longrightarrow E' \\
\mathsf{let}\ X : U = C\ \mathsf{in}\ T_1 & \longrightarrow [\![C/X]\!]T_2 \\
\mathsf{let}\ X_1 : U_1 = (\mathsf{let}\ X_2 : U_2 = E_1\ \mathsf{in}\ E_2)\ \mathsf{in}\ T_3 & \longrightarrow \mathsf{let}\ X_2 : U_2 = E_1\ \mathsf{in}\ \mathsf{let}\ X_1 : U_1 = E_2\ \mathsf{in}\ T_3 \\
\mathsf{let}\ X : U = E\ \mathsf{in}\ T & \longrightarrow T \qquad \text{if } X \notin \mathsf{FMV}(T)
\end{array}$$

$\boxed{E \longrightarrow E'}$ Expression E reduces to E', the contextual closure of the following relation

$$\begin{array}{rl}
(\mathsf{fn}\ x.\ E_1)\ E_2 & \longrightarrow [E_2/x]E_1 \\
(\Lambda X.\ E_1)\ C & \longrightarrow [\![C/X]\!]E_1 \\
\mathsf{rec}\ f.E & \longrightarrow [\mathsf{rec}\ f.E/f]E \\
\mathsf{case}\ C\ \mathsf{of}\ \vec{B} & \longrightarrow [\![\theta]\!]E_i \text{ where } \Delta_i\ .\ C_i \mapsto E_i \in \vec{B} \\
 & \qquad \text{and } \Delta, \Delta_i \vdash C \doteq C_i/(\theta, \Delta) \\
\mathsf{let}\ X_1 : U_1 = (\mathsf{let}\ X_2 : U_2 = E_1\ \mathsf{in}\ E_2)\ \mathsf{in}\ E_3 & \longrightarrow \mathsf{let}\ X_2 : U_2 = E_1\ \mathsf{in}\ \mathsf{let}\ X_1 : U_1 = E_2\ \mathsf{in}\ E_3 \\
\mathsf{let}\ X : U = E_1\ \mathsf{in}\ E_2 & \longrightarrow E_2 \qquad \text{if } X \notin \mathsf{FMV}(E_2)
\end{array}$$

Fig. 1. Reduction rules. Expressions assumed to live in ambient meta context Δ.

form, so type equivalence is defined modulo computation. In general, types need not be closed, and in the presence of binding constructs in types, this means expressions appearing in types need not be closed. Thus case constructs can get "stuck" during evaluation (when the scrutinee is non-ground). Therefore we reason about expression equivalence not just modulo β-reduction, but further strengthen the expression equivalence relation to include the same equational theory as for types, observing that case analysis constructs are a generalization of the "let-in" construct, and therefore commute just as "let-in" constructs do. In meta context Δ and context Γ,

$\mathsf{let}\ X : U = E_1\ \mathsf{in}\ E_2$ is syntactic sugar for $\mathsf{case}\ E_1\ \mathsf{of}\ \Delta, X{:}U\ .\ X \mapsto E_2$.

The rules for type and expression reduction are given in Figure 1. They include the usual β-reduction rules. In addition, we have non computational reduction rules, called *commuting conversions*, or π-reduction, that help us identify types and terms up to associativity of "let-in" [1].

3.2 The abstract typing relation

Armed with a notion of equivalence between types, we can now formulate a type system for the computational language. It is described formally in Figure 2. The typing of atomic expressions is straightforward; we either have a variable or a meta term, in which case we type check the meta term according to the rules given in [8, 5]. Applying a function to a meta term requires substituting the meta-term in the result type. One essential characteristics of this type system is that, contrary to existing two-level systems, which to our knowledge always only identify types up to syntactic equality, here a type T_1 can be substituted for a type T_2 so long as a rather stronger notion of equivalence holds between T_1 and T_2. Type equivalence is defined in Figure 3.

$\boxed{\Delta \vdash \Gamma\ \mathsf{ctx}}$ Context Γ is well-formed

$$\frac{\vdash \Delta\ \mathsf{mctx}}{\Delta \vdash \cdot\ \mathsf{ctx}} \qquad \frac{\Delta; \Gamma \vdash T\ \mathsf{ctype} \quad \Delta \vdash \Gamma\ \mathsf{ctx}}{\Delta \vdash \Gamma, x{:}T\ \mathsf{ctx}} \qquad \frac{\Delta; \Gamma \vdash E : U \quad \Delta \vdash C : U \quad \Delta \vdash \Gamma\ \mathsf{ctx}}{\Delta \vdash \Gamma, E \rightsquigarrow C : U\ \mathsf{ctx}}$$

$\boxed{\Delta \vdash T\ \mathsf{ctype}}$ Computational type T is well-formed

$$\frac{\Delta \vdash T_1\ \mathsf{ctype} \quad \Delta \vdash T_2\ \mathsf{ctype}}{\Delta \vdash T_1 \to T_2\ \mathsf{ctype}} \qquad \frac{\Delta, X{:}U \vdash T\ \mathsf{ctype}}{\Delta \vdash \Pi X{:}U.T}$$

$$\frac{\vdash \Delta\ \mathsf{mctx} \quad \Delta \vdash U\ \mathsf{mtype}}{\Delta \vdash U\ \mathsf{ctype}} \qquad \frac{\Delta; \cdot \vdash E : U \quad \Delta, X{:}U \vdash T\ \mathsf{ctype}}{\Delta \vdash \mathsf{let}\ X : U = E\ \mathsf{in}\ T}$$

$\boxed{\Delta; \Gamma \vdash E : T}$ Computational expression E has type T

$$\frac{\vdash \Delta\ \mathsf{mctx} \quad \Delta \vdash \Gamma\ \mathsf{ctx} \quad \Gamma(x) = T}{\Delta; \Gamma \vdash x : T} \qquad \frac{\vdash \Delta\ \mathsf{mctx} \quad \vdash \Gamma\ \mathsf{ctx} \quad \Delta \vdash C : U}{\Delta; \Gamma \vdash C : U}$$

$$\frac{\Delta; \Gamma \vdash E_1 : T_1 \to T_2 \quad \Delta; \Gamma \vdash E_2 : T_1}{\Delta; \Gamma \vdash E_1\ E_2 : T_2} \qquad \frac{\Delta; \Gamma \vdash E : \Pi X{:}U.T \quad \Delta; \Gamma \vdash C : U}{\Delta; \Gamma \vdash E\ C : [\![C/X]\!]T}$$

$$\frac{\Delta; \Gamma \vdash E : T_1 \quad \Delta; \Gamma \vdash T_2\ \mathsf{ctype} \quad \Delta; \Gamma \vdash T_1 \equiv T_2}{\Delta; \Gamma \vdash E : T_2}$$

$$\frac{\Delta; \Gamma, y{:}T_1 \vdash E : T_2}{\Delta; \Gamma \vdash \mathsf{fn}\ y.\ E : T_1 \to T_2} \qquad \frac{\Delta; \Gamma,\ f : T \vdash E : T}{\Delta; \Gamma \vdash \mathsf{rec}\ f.E : T} \qquad \frac{\Delta, X{:}U; \Gamma \vdash E : T}{\Delta; \Gamma \vdash \Lambda X.\ E : \Pi X{:}U.T}$$

$$\frac{\Delta; \Gamma \vdash E : U \quad \text{for all } i\ \Delta; \Gamma \vdash B_i :^{E}_{U} T}{\Delta; \Gamma \vdash \mathsf{case}\ E\ \mathsf{of}\ \vec{B} : T}$$

$\boxed{\Delta; \Gamma \vdash B :^{E}_{U} T}$ Branch B with scrutinee E of type U has type T

$$\frac{\vdash \Delta_i\ \mathsf{mctx} \quad \Delta_i \vdash C : U_i \quad \Delta, \Delta_i, U_i = U_s; \Gamma, E_s \rightsquigarrow C : U_s \vdash E : T}{\Delta; \Gamma \vdash \Delta_i\ .\ C \mapsto E :^{E_s}_{U_s} T}$$

Fig. 2. Abstract typing relation.

The other essential characteristic is that we record any information that we may learn during case analysis in *equality assumptions*. Adding new equality assumptions to the context or meta context makes more terms equivalent. Case analysis can be done on suitably raised meta terms at base type. But since we permit meta terms to be dependently typed, assuming that the pattern in some given branch matches the scrutinee implies that the type of the scrutinee and the type of the pattern are unifiable. Moreover, we also learn that the pattern and the scrutinee must be equivalent to some instance of the pattern. The first equality assumption is a fact about how type indices of data level entities are related; the latter tells us something about the result of proof level computations. Both kinds of information, according to the rules of Figure 3, can be exploited to decide whether two types are convertible.

Given two syntactically distinct types (or expressions), which we generically denote as Z_1, Z_2, we might get one step closer to showing their equivalence by reducing either of them, or rewriting them using the equality assumptions at

$\boxed{\Delta; \Gamma \vdash Z_1 \equiv Z_2}$ Type and expression equivalence rules

$$\frac{Z_1 \longrightarrow Z_1' \quad \Delta; \Gamma \vdash Z_1' \equiv Z_2}{\Delta; \Gamma \vdash Z_1 \equiv Z_2} \qquad \frac{}{\Delta; \Gamma \vdash Z_1 \equiv Z_1} \qquad \frac{Z_2 \longrightarrow Z_2' \quad \Delta; \Gamma \vdash Z_1 \equiv Z_2'}{\Delta; \Gamma \vdash Z_1 \equiv Z_2}$$

$$\frac{U_1 = U_2 \in \Delta \quad \Delta \vdash U_1 \doteq U_2/(\theta, \Delta') \quad \Delta'; [\![\theta]\!]\Gamma \vdash [\![\theta]\!]Z_1 \equiv [\![\theta]\!]Z_2}{\Delta; \Gamma \vdash Z_1 \equiv Z_2}$$

$$\frac{E \rightsquigarrow C : U \in \Gamma \quad \mathcal{C} \in \mathsf{split}(Z_1, E) \quad \Delta; \Gamma \vdash [\![C/X]\!]\mathcal{C}[\mathsf{id}(X)] \equiv Z_2}{\Delta; \Gamma \vdash Z_1 \equiv Z_2}$$

$$\frac{E \rightsquigarrow C : U \in \Gamma \quad \mathcal{C} \in \mathsf{split}(Z_2, E) \quad \Delta; \Gamma \vdash Z_1 \equiv [\![C/X]\!]\mathcal{C}[\mathsf{id}(X)]}{\Delta; \Gamma \vdash Z_1 \equiv Z_2}$$

Fig. 3. Type and expression equivalence rules

our disposal. Dynamically, ground instances of the type of a scrutinee of a case analysis must match the type of the pattern, hence the two must unify. We can exploit this statically by finding the most general unifier and instantiating the free meta variables in Z_1, Z_2 accordingly. The last alternative is to rewrite any occurrence of the scrutinee E of a case analysis with the pattern C against which the value of scrutinee is assumed to match. We express this by non determistically *splitting* the type or expression Z into a reduction context $\mathcal{C}$, such that *plugging* E into $\mathcal{C}$ yields the original, *i.e.* $\mathcal{C}[E] = Z$. Given that a value of E matches C, we can plug C instead. However, $\mathcal{C}$ may need to be renamed appropriately to avoid captures. Since the base theory already provides us with a notion of capture avoiding substitution of meta terms, we instead plug a fresh variable X, for which we finally substitute C.

In dependently typed systems with eliminators, such as the Calculus of Inductive Constructions or Martin-Löf Type Theory, just how the information gained from a case analysis is used is determined by a user supplied function that explains how the target type of the whole case analysis should be refined in each branch. As noted in [9], however, storing equality assumptions instead is more flexible: in each branch, refinements can also occur in the context rather than just in the target type T of a judgement $\Delta; \Gamma \vdash \mathsf{case}\ E\ \mathsf{of}\ \vec{B} : T$. This obviates the need for the awkward *convoy pattern* commonly seen in Coq, where users discharge select assumptions from the context Γ into T just before a case analysis, in order for refinement to occur in the right places.

We chose to make "let-in" a special case of a case analysis. But in metatheoretic arguments, it may be more convenient to use a more specialized typing rule.

Theorem 1. *The following typing rule is admissible:*

$$\frac{\Delta; \Gamma \vdash E_1 : U \quad \Delta, X{:}U; \Gamma, E_1 = \mathsf{id}(X) \vdash E_2 : T}{\Delta; \Gamma \vdash \mathsf{let}\ X : U = E_1\ \mathsf{in}\ E_2 : T}$$

Example 2. The following expression cannot be typed against the given type:

```
M : [.nat]; · ⊬ fn x ⇒ x : let [.N] = plus [.M] [.M] in
                              [.vector N] → [.vector N]
```

However, doing a case analysis beforehand gets us out of this particular rut:

```
M : [.nat]; · ⊢ let [.N'] = plus [.M] [.M] in fn x ⇒ x
              : let [.N] = plus [.M] [.M] in [.vector N] → [.vector N]
```

The derivation (left as an exercice to the reader) uses the fact that inside the branch of the case analysis, we have extra information available that lets us reduce away the enclosing "let-in" in the type. This example shows that terms must contain *evidence* showing how to eliminate irreducible forms in types.

The abstract typing relation is suitable for a metatheoretical study, but one shortcoming is that this relation does not readily inform us of a type checking procedure given a meta context, context, term and type as input. The major issue is that this relation does not commit to particular strategy as to when and where to use the conversion rule. We study some of the metatheory of this system in Section 4. In the long version of this paper, we show a bidirectional type checking algorithm that we prove sound and complete with respect to the rules given in this section. The construction of such an algorithm follows the same lines as [10] — as for theirs, completeness relies on standardization of weak head reduction [11].

4 Metatheoretical properties

The interested reader may find the proofs for the properties in this section in the long version of this paper. The type system presented in Section 3 enjoys the usual structural properties, such as weakening and substitution. To these standard structural properties we add that equality assumptions can be permuted.

Lemma 3 (Permutation).

1. *If* $\Delta_1, U_1 = U_2, U_1' = U_2', \Delta_2; \Gamma \vdash J$ *then* $\Delta_1, U_1' = U_2', U_1 = U_2, \Delta_2; \Gamma \vdash J$*;*
2. *if* $\Delta; \Gamma_1, E_1 \rightsquigarrow C_1 : U_1, E_2 \rightsquigarrow C_2 : U_2, \Gamma_2 \vdash J$ *then* $\Delta; \Gamma_1, E_2 \rightsquigarrow C_2 : U_2, E_1 \rightsquigarrow C_1 : U_1, \Gamma_2 \vdash J$*.*

Using typing assumptions can be delicate, in that it must be taken into account that the type conversion rule can be used at any point. An important but straightforward result is the use of the conversion rule can always be stripped out of the root of the derivation.

Lemma 4 (Inversion).

1. *If* $\Delta; \Gamma \vdash x : T$ *then* $T \equiv \Gamma(x)$*;*
2. *if* $\Delta; \Gamma \vdash C : T$ *then* $T \equiv U$ *for some* U*;*
3. *if* $\Delta; \Gamma \vdash E_1\ E_2 : T$ *then* $\Delta; \Gamma \vdash E_1 : T_1 \to T_2$ *and* $\Delta; \Gamma \vdash E_2 : T_1$ *and* $T \equiv T_2$ *for some* T_1, T_2*;*
4. *if* $\Delta; \Gamma \vdash E_1\ C : T$ *then* $\Delta; \Gamma \vdash E_1 : \Pi X{:}U.T_2$ *and* $\Delta; \Gamma \vdash C \Leftarrow U$ *and* $T \equiv [\![C/X]\!]T_2$ *for some* U, T_2*;*

5. *if* $\Delta; \Gamma \vdash \mathsf{fn}\ x.\, E : T$ *then* $T \equiv T_1 \to T_2$ *for some* T_1, T_2*;*
6. *if* $\Delta; \Gamma \vdash \Lambda X.\, E : T$ *then* $T \equiv \Pi X{:}U.T_2$ *for some* U, T_2*;*
7. *if* $\Delta; \Gamma \vdash \mathsf{case}\ E\ \mathsf{of}\ \vec{B} : T$ *then* $\Delta; \Gamma \vdash E : U$ *and for all* i $\Delta; \Gamma \vdash \vec{B}_i :^E_U T'$ *and* $T \equiv T'$ *for some* U, T'*.*

From then on, we can prove a number of important properties about equalities assumptions. The following two say that the left and right hand sides of an equality assumption are equivalent, then the assumption is not informative and so might as well be done without. Moreover, equality assumptions can be decomposed into simpler ones.

Lemma 5 (Cut).

1. *If* $\Delta_1, U_1 = U_2, \Delta_2; \Gamma \vdash J$ *and* $\Delta_1; \Gamma \vdash U_1 \equiv U_2$*, then* $\Delta_1, \Delta_2; \Gamma \vdash J$*;*
2. *if* $\Delta; \Gamma_1, E \rightsquigarrow C : U, \Gamma_2 \vdash J$ *and* $\Delta; \Gamma_1 \vdash E \equiv C$*, then* $\Delta; \Gamma_1, \Gamma_2 \vdash J$*.*

Proof. The first part can be proved by structural induction the first derivation and inversion on the equivalence assumption. By structural induction on first derivation. The second part can be proved by double induction on the first derivation and the equivalence relation. □

Lemma 6.
If $\Delta; \Gamma, (\mathsf{let}\ X : U = E_1\ \mathsf{in}\ E_2) = C \vdash J$
then $\Delta, X{:}U; \Gamma, E_1 = \mathsf{id}(X), E_2 = C \vdash J$*.*

We are now in a position to prove the first half of type safety: that types are invariant under reduction of expressions. The above two lemmas are useful for the case where case analyses are reduced and for the permutation rules.

Theorem 7 (Preservation). *If* $\Delta; \cdot \vdash E : T$ *and* $E \longrightarrow E'$ *then* $\Delta; \cdot \vdash E' : T$*.*

If we restrict our attention to only closed forms, then we can state the following preservation lemma. In our case, the only values are functions and meta terms. Of course, for progress to be true we need to assume that pattern matching never gets "stuck", *i.e.* that all case analyses coverage check [12].

Lemma 8 (Canonical forms).

1. *If* $\Delta; \Gamma \vdash V : U$ *then* V *is of the form* C*;*
2. *if* $\Delta; \Gamma \vdash V : T_1 \to T_2$ *then* V *is of the form* $\mathsf{fn}\ x.\, E$*;*
3. *if* $\Delta; \Gamma \vdash V : \Pi X{:}U.T$ *then* V *is of the form* $\Lambda X.\, E$*.*

Lemma 9. *If* $\cdot; \cdot \vdash E : T$ *and* E *coverage checks, then either* E *is a value or* $E \longrightarrow E'$ *for some* E'*.*

Lemma 10 (Type safety). *If* $\cdot; \cdot \vdash E : T$ *then either there exsists a* V *such that* $E \longrightarrow^* V$*, or* E *diverges.*

The computations in the proof level would not be meaningful proofs if these computations were not total. We do not attempt to address termination and coverage issues here — there are a number of ways of restricting computations to only terminating ones, using simple syntactic guard conditions or more semantic methods such as sized types. Rather than committing to any particular termination scheme, we show that adding permutation rules does not affect normalization, by embedding expressions of our system into a simply typed variant of the computational language by means of a CPS translation, where type indices are erased (index erasing is denoted $(\cdot)^-$) and permutations are simulated by β-reduction.

Theorem 11. *If* $\Delta^-; \Gamma^- \vdash E^- : T^-$ *and* $\mathcal{SN}_\beta(E)$ *then* $\mathcal{SN}_{\beta\pi}(E)$.

By embedding types into expressions, we can lift this result to types.

5 Proofs by reflection

Given a formula φ under hypotheses Γ expressed in a consistent formal deduction system $\mathcal{D}$, if it is provable in $\mathcal{D}$ then one can construct a cut-free derivation justifying the judgement $\Gamma \vdash \varphi$. But cut-free proofs can be really quite large. Besides, given a family of similar formulas (φ_k) over a decidable theory $\mathcal{T}$, each formula φ_i will in general have a completely different cut-free proof. It can be tedious for the user to have to write out such long proofs, especially if the problem domain is small and easily automated.

An alternative, especially if the family (φ_k) is well characterized, such as the set of ring inequalities or tautologies of propositional logic, is to rely on the answer of decision procedure f for a particular φ_i, rather than manually proving it. But this solution requires to trust f, hence increasing the trusted base. If one can instead implement f as a term of discourse, then f can be reasoned about, with the view towards eliminating f from the trusted base. In particular, if one can show the following soundness lemma about f,

$$\mathsf{soundness} : \forall k, f\ k = \mathsf{true} \to \varphi_k$$

then for any φ_i, by the above lemma we need only prove that $f\ i$ computes to true to prove φ_i. In dependently typed systems, types are usually equated modulo some fixed notion of computation. Therefore, if we can implement f as a closed function of the term language, such that $f\ i = \mathsf{true}$ holds definitionally, then $f\ i = \mathsf{true}$ can be established by reflexivity of equality alone, so that the following is a proof of say (φ_1):

$$\mathsf{soundness}\ 1\ (\mathsf{refl}\ (f\ 1)\ \mathsf{true})$$

This proof is emphatically not cut-free. It is typically much shorter than any cut-free proof of φ_1 could ever be. We trade away proof size against more computation during proof checking. But even more importantly, for any φ_i that P can prove, the proof is of exactly the same shape as for every other formula of this family of formulas. The only varying parameter in each of these proofs is the number i.

It is, however, often completely impractical to identify a formula by an integer i, in effect a Gödel number. The efficacy of this proof technique hinges upon having a way to represent formulas more conveniently than with an integer, say as inductively constructed data, *i.e.* to *reflect* the language of terms within itself. Mapping from formulas to their representation is called *quoting* (or *metaification*), which we write $\ulcorner\cdot\urcorner$. The difficulty is that in general φ_i may involve free variables, or include binding constructs, which must be represented somehow — CMTT provides just such convenient, well-behaved, adequate representations.

Proofs by reflection are an important and practically useful proof methodology to have in one's toolbox, and has seen wide adoption in dependently typed interactive proof assistants such as in Coq [13–18], in Agda under the guise of *universes* [19] and even in non-dependently typed proof assistants such as HOL and Isabelle, where they are called *pro-forma theorems* [20]. In every case, one difficulty lies in constructing the appropriate representation for each formula one seeks to prove by reflection, *i.e.* how to perform quotation. The core language of these systems don't offer any kind of primitive support for quotation, which must therefore be implemented at the meta-level. In tactic based systems, quotation can be implemented as a tactic. [21] propose to leverage the support in the elaborator of Coq for *unification hints* (also available in Matita) to construct representations automatically and declaratively, rather than through an opaque tactic that can bear no formal reasoning about it within Coq itself. In systems with neither tactics nor unification hints, representations must be constructed by hand, independently for each goal — an implementation technique which obviously doesn't scale. Either way, one must also prove that unquoting each representation yields the original formula.

Our framework, which build on earlier work [6, 5, 8] on CMTT, obviates the need for a custom quoting and interpretation function, because the language provides a primitive notion of quotation through pattern matching on (open) proofs and formulas. We therefore do not need to provide custom interpretation functions either, or prove that representations map to their respective formulas. What's more, we gain powerful reasoning principles that permits encodings of data as higher order abstract syntax (HOAS), hence making binding, scope and substitution much easier to deal with.

We demonstrate this through two examples: deciding monoidal equalities, such as it appears in [13], and a normalizer for HOAS encoded λ-terms.

5.1 Example 1: loop simplification

We demonstrate in this section a simple decision procedure for a class of equalities that frequently occur when reasoning about numbers, lists, and any other structure that features an identity element and an associative binary operator.

Naturals form a *loop* (a monoid without associativity) under addition, since the following laws hold:

1. $\forall n.\ 0 + n = n$ (left identity);
2. $\forall n.\ n + 0 = n$ (right identity);

It is tedious to have to prove equalities involving only 0 and addition, *e.g.* during the reversal of a length indexed vector. Given two natural number expressions, we can however find a canonical representative of the equivalence class of each under the above equational theory, and compare the canonical representatives for syntactic equality to determine whether the two are provably equal. In his seminal paper on reflection [13], Boutin gives the following normalization function to find canonical representatives (transposed into Beluga syntax):

```
rec norm : [g. nat] → [g. nat] =
  fn m ⇒case m of
    | [g. add M1.. M2..] ⇒(case norm [g. M1..], norm [g. M2..] of
        | [g. zero], [g. M2'..] ⇒[g. M2'..]
        | [g. M1'..], [g. zero] ⇒[g. M1'..]
        | [g. M1'..], [g. M2'..] ⇒[g. add (M1'..) (M2'..)])
    | [g. _] ⇒m;
```

We can show soundness of this normalization function, in the sense that any output is always related to the input under the above equivalence relation,

```
soundness : {M : [g. nat]} let [g.N..] = norm [g.M..]in [g.eq (M..) (N..)];
```

by induction on `[g. M..]`. We can decide whether two number expressions are equal by normalizing both sides and comparing:

```
rec decide : [g. nat] → [g. nat] → [. bool] = fn m1 ⇒fn m2 ⇒
    if normalize m1 == normalize m2 then [. true] else [. false];
```

where `==` is a primitive computation level syntactic equality test. We can show the fundamental reflection lemma about `decide`, which says that it is a sound decision procedure,

```
reflect : let [. B] = decide [g. M1..] [g. M2..] in
            [. eqb B true] → [g. eq (M1..) (M2..)];
```

which follows from the soundness result above. Now if we have a concrete expression `m1 = [x,y. (mplus x (mplus y mzero)]` and `m1 = [x,y. (mplus x (mplus mzero y)]`, then the following is a proof of their equality:

```
reflect m1 m2 [. eqb/refl];
```

The full code for this example is given in Appendix A.

5.2 Example 2: higher order term equality

Similarly, we can, more ambitiously, decide convertibility of terms that contain binding structures, such as the pure λ-calculus that we defined in Section 1. We can write a normalization function on terms, `norm : (g : ctx)[g. ι] → [g. ι]`. This function is obviously partial: not every λ-term has a normal form. But we may still prove, *e.g.* by constructing `norm` using normalization by evaluation [22, 7], that this normalization function is sound with respect to iterated reduction:

```
soundness  : let [g.M'..] = norm [g.M..] in [g. red∗ (M..) (M'..)];
```

If `convertible` is the symmetric closure of `red`$*$, given two concrete λ-terms `m1` and `m2`, we can prove the formula

```
let [g. M1..] = m1 in let [g. M2..] = m2 in [g. convertible (M1..) (M2..)];
```

using a reflection lemma, as in Section 5.1. Reflection on propositions involving terms with binders is where our proposal really shines: the `soundness` theorem is non trivial to establish. However, because it is defined over HOAS representations, we can use a number of properties for free, such as substitution lemmas and static guarantees of well scoping. Proofs by reflection with binders is just as convenient as in the first order case.

6 Related work

Beyond the support for proofs by reflection in other systems already discussed in Section 5, this work draws on ideas from a variety of proof environments. In particular, a variety of systems have emerged in recent years to offer first class support for rich representations of syntax. Our works builds on contextual LF [5], from which we inherit a rich computational language supporting case analysis on data level terms as well as on contexts. In the style of [7], we remain generic in the language of the data level. LF is but one instantiation — other choices could include instantiating the data level language with the computational language itself. In the style of [23, 24], we draw type dependencies from a language distinct from that of the computational language. Other closely related systems designed around two levels include Delphin [25], VeriML [26, 27], which all have in common the manipulation of contextual objects. However, these systems maintain a strict separation of levels. In particular, one cannot include computations as the domain of discourse. Licata and Harper [28] present a library within Agda where mixing between computational and data languages is allowed. However, theirs is a simply typed universe. Also, in their system, mixing means that certain structural properties such as weakening or substitution do not hold in general.

Permuting conversions have been considered in a number of works. The idea of translating away the conversions using a CPS translation can be traced back to de Groote [29]. Various calculi with sums with or without extensionality axioms have been developed over the years (see [30, 31]). One particularly interesting point is how to exploit dynamic assumptions about case analyses statically. One closely related work in this regard is [9], who also choose to capture information gained during case analysis as equality assumptions added to the context. Our setting is simpler, because their equivalence relation on expressions is an arbitrary relation, whereas ours is inductively defined, and so only captures equivalence of terms that can be shown to be such in a finite number of steps. This removes a number of technical difficulties in the design. Also, our dependent product abstracts only over meta terms, not computations, so the properties required to show preservation are weaker. Nonetheless, the design of our equivalence relation owes much to theirs. Balat et al [32] show a normalization by evaluation based algorithm to decide the equational theory of terms including strong sums. Generalizing their algorithm to our setting would afford us a stronger notion of type equivalence.

7 Conclusion

The impetus for this work was the observation that CMTT, through its lifting of terms to meta terms, *i.e.* (open) objects with observable structure, already provides first class support for one of the more delicate parts of a proof by reflection: quoting. The precise design and capabilities of the computational language layered on top is immaterial, but for the ability to reflect upon entities of the computational language. We have shown in this paper that such reflection can be achieved with a very lightweight extension to the types of the computational language. To support convenient metareasoning, we extended the equational theory of types and expressions with standard extensional axioms, effectively viewing meta objects as objects of a "quoting" monad. But our extensional principles are still weak: our language is one of *weak sums* as opposed to *strong* (or *categorical*) *sums*. We could envisage commuting not just the computationally uninformative "let", but also the destructuring and multi-branch "case". Such axioms would correspond to η laws for expressions of base type. Moreover, the use of constraints during case analysis rather than substitution paves the way for including more esoteric permutation rules, such as commuting function application with case analysis, even in in the presence of dependent types such as here.

Acknowledgments: Many thanks to Brigitte Pientka for many fruitful discussions that greatly influenced this work.

References

1. Moggi, E.: Computational lambda-calculus and monads. In: LICS, IEEE Computer Society (1989) 14–23
2. Pientka, B., Dunfield, J.: Beluga: a framework for programming and reasoning with deductive systems (System Description). In Giesl, J., Haehnle, R., eds.: 5th International Joint Conference on Automated Reasoning (IJCAR'10). Lecture Notes in Artificial Intelligence (LNAI 6173), Springer-Verlag (2010) 15–21
3. Harrison, J.: Hol light: An overview. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M., eds.: TPHOLs. Volume 5674 of Lecture Notes in Computer Science., Springer (2009) 60–66
4. Watkins, K., Cervesato, I., Pfenning, F., Walker, D.: A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University (2002)
5. Pientka, B.: A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In: 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08), ACM Press (2008) 371–382
6. Pientka, B., Dunfield, J.: Programming with proofs and explicit contexts. In: ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08), ACM Press (July 2008) 163–173
7. Cave, A., Pientka, B.: Programming with binders and indexed data-types. [33] 413–424
8. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Transactions on Computational Logic **9**(3) (2008) 1–49

9. Jia, L., Zhao, J., Sjöberg, V., Weirich, S.: Dependent types and program equivalence. In Hermenegildo, M.V., Palsberg, J., eds.: POPL, ACM (2010) 275–286
10. Abel, A., Altenkirch, T.: A partial type checking algorithm for type: Type. Electr. Notes Theor. Comput. Sci. **229**(5) (2011) 3–17
11. Plotkin, G.: Call-by-name, call-by-value and the λ-calculus. Theoretical Computer Science **1**(2) (1975) 125–159
12. Pientka, B., Dunfield, J.: Covering all bases: design and implementation of case analysis for contextual objects. Technical report, McGill University (2010)
13. Boutin, S.: Using reflection to build efficient and certified decision procedures. In Abadi, M., Ito, T., eds.: TACS. Volume 1281 of Lecture Notes in Computer Science., Springer (1997) 515–529
14. Gonthier, G.: The four colour theorem: Engineering of a formal proof. In Kapur, D., ed.: ASCM. Volume 5081 of Lecture Notes in Computer Science., Springer (2007) 333
15. Grégoire, B., Mahboubi, A.: Proving equalities in a commutative ring done right in coq. In Hurd, J., Melham, T.F., eds.: TPHOLs. Volume 3603 of Lecture Notes in Computer Science., Springer (2005) 98–113
16. Verma, K.N., Goubault-Larrecq, J., Prasad, S., Arun-Kumar, S.: Reflecting bdds in coq. In He, J., Sato, M., eds.: ASIAN. Volume 1961 of LNCS., Springer (2000) 162–181
17. Théry, L.: Proof pearl: Revisiting the mini-rubik in coq. In Mohamed, O.A., Muñoz, C., Tahar, S., eds.: TPHOLs. Volume 5170 of Lecture Notes in Computer Science., Springer (2008) 310–319
18. Gonthier, G., Mahboubi, A., Tassi, E.: A Small Scale Reflection Extension for the Coq system. Technical report RR-6455, INRIA (2008)
19. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda—a functional language with dependent types. In: 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09). Volume 5674 of Lecture Notes in Computer Science., Springer (2009) 73–78
20. Harrison, J.: Metatheory and reflection in theorem proving: A survey and critique. Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK (1995)
21. Gonthier, G., Ziliani, B., Nanevski, A., Dreyer, D.: How to make ad hoc proof automation less ad hoc. In Chakravarty, M.M.T., Hu, Z., Danvy, O., eds.: ICFP, ACM (2011) 163–175
22. Berger, U., Schwichtenberg, H.: An inverse of the evaluation functional for typed lambda-calculus. In: Logic in Computer Science. (1991) 203–211
23. Zenger, C.: Indexed types. Theoretical Computer Science **187**(1-2) (1997) 147–165
24. Xi, H., Pfenning, F.: Dependent types in practical programming. In: 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99), ACM Press (1999) 214–227
25. Poswolsky, A.B.: Functional Programming with Logical Frameworks: The Delphin Project. CreateSpace, Paramount, CA (2008)
26. Stampoulis, A., Shao, Z.: VeriML: typed computation of logical terms inside a language with effects. In Hudak, P., Weirich, S., eds.: 15th ACM SIGPLAN International Conference on Functional Programming (ICFP'10), ACM (2010) 333–344
27. Stampoulis, A., Shao, Z.: Static and user-extensible proof checking. [33] 273–284
28. Licata, D.R., Harper, R.: A universe of binding and computation. In Hutton, G., Tolmach, A.P., eds.: 14th ACM SIGPLAN International Conference on Functional Programming, ACM Press (2009) 123–134
29. de Groote, P.: A cps-translation of the lambda-μ-calculus. In Tison, S., ed.: CAAP. Volume 787 of Lecture Notes in Computer Science., Springer (1994) 85–99

30. Cosmo, R.D., Kesner, D.: A confluent reduction for the extensional typed lambda-calculus with pairs, sums, recursion and terminal object. In Lingas, A., Karlsson, R.G., Carlsson, S., eds.: ICALP. Volume 700 of Lecture Notes in Computer Science., Springer (1993) 645–656
31. Dougherty, D.J.: Some lambda calculi with categorial sums and products. In Kirchner, C., ed.: RTA. Volume 690 of Lecture Notes in Computer Science., Springer (1993) 137–151
32. Balat, V., Cosmo, R.D., Fiore, M.P.: Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In Jones, N.D., Leroy, X., eds.: POPL, ACM (2004) 64–76
33. Field, J., Hicks, M., eds.: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012. In Field, J., Hicks, M., eds.: POPL, ACM (2012)

A Full code for Example 1

```
bool  : type.
true  : bool.
false  : bool.

bottom  : type.

monoid  : type.
mzero  : monoid.
mplus  : monoid  → monoid  → monoid.

eq  : monoid  → monoid  → type.
eq/refl  : eq M M.

associative  : eq (mplus (mplus M1 M2) M3) (mplus M1 (mplus M2 M3)).
neutral_right  : eq (mplus M1 mzero) M1.
neutral_left  : eq (mplus mzero M1) M1.

eqb  : bool  → bool  → type.
eqb/refl  : eq B B.

schema ctx = monoid;

rec normalize  : [g. monoid]  → [g. monoid] =
    fn m ⇒case m of
      | [g. mplus M1.. M2..] ⇒case normalize [g. M1..], normalize [g.
          M2..] of
          | [g. mzero], [g. M2'..] ⇒[g. M2'..]
          | [g. M1'..], [g. mzero] ⇒[g. M1'..]
          | [g. M1'..], [g. M2'..] ⇒[g. mplus M1'.. M2'..]
      | [g. _] ⇒m
;

% Note :  Beluga does not check termination of this function, which
% would need be justified by the fact that normalize does not increase
% the size of a term.
rec correctness  : {M : [g. monoid]}
                  let [g. M'] = normalize [g. M..] in [g. eq M M'] =
```

```
    λM ⇒case [g. M..] of
      | [g. #p] ⇒[g. eq/refl]
      | [g. mzero] ⇒[g. eq/refl]
      | [g. mplus M1.. M2..] ⇒
        let eqH1 = correctness [g. M1..] in
        let eqH2 = correctness [g. M2..] in
        case normalize [g. M1..], normalize [g. M2..] of
           | [g. mzero], [g. M2'..] ⇒(case eqH2 of
               [g. eq/refl] ⇒[g. neutral_left M2'..])
           | [g. M1'..], [g. mzero] ⇒(case eqH1 of
               [g. eq/refl] ⇒[g. neutral_right M1'..])
           | [g. M1'..], [g. M2'..] ⇒(case eqH1, eqH2 of
               [g. eq/refl], [g. eq/refl] ⇒[g. mplus M1'.. M2'..])
;

rec decide  : [g. monoid]  → [g. monoid]  → [. bool] =
    fn m1 ⇒fn m2 ⇒
    let v1 = normalize m1 in
    let v2 = normalize m2 in
    if v1 == v2 then [. true] else [. false]
;

rec reflect  : {M1 : [g. monoid]} {M2 : [g. monoid]}
               let [. B] = decide [g. M1..] [g. M2..] in [. eqb B true]
                     → [g. eq (M1..) (M2..)] =
    λM1 ⇒λM2 ⇒
    (let [g. M1'..] = normalize [g. M1..] in
     let [g. M2'..] = normalize [g. M2..] in
     let eqH1 = correctness [g. M1..] in
     let eqH2 = correctness [g. M2..] in
     case eqH1, eqH2 of
       | [g. eq/refl], [g. eq/refl] ⇒case [g. M1'..] == [g. M2'..] of
       | True ⇒fn eqbH ⇒[g. eq/refl]
       | False ⇒fn eqbH ⇒impossible)

let t1 = let m1 = [x,y. (mplus x (mplus y mzero)] in
         let m2 = [x,y. (mplus x (mplus mzero y)] in
         reflect m1 m2 [. eqb/refl];
```