Categories and Monads in HOL-Omega

Jeremy E. Dawson *

Logic and Computation Group,
College of Engineering and Computer Science,
Australian National University, Canberra, ACT 0200, Australia
`http://cecs.anu.edu.au/~jeremy/`

Abstract. We consider HOL-Omega, a recent extension by Homeier to the type system of the HOL theorem prover. We describe how it permits an implementation of categories, where we model the objects of a category as the types of HOL-Omega. This gives a "light-weight" implementation of categories into a theorem prover: we explore how this is sufficient for easily proving some non-trivial results of category theory, although some other results cannot be expressed in HOL-Omega in this way. We illustrate the use of HOL-Omega by first proving the multiple characterisations of an adjoint pair of functors. Then we consider compound monads, where we consider distributive laws and the conditions for compatibility of monads. We show how in HOL-Omega we can express the concept of a monad in the Kleisli category of another monad, and use this to obtain easy proof of some results about distributive laws, including a new result simplifying the conditions of Barr & Wells for compatibility of monads.

Keywords: formal verification, distributive law for monads, Kleisli category

1 Introduction

Monads and Adjoint Functors Monads have long been known in category theory, where a monad consists of a endofunctor with two natural transformations η and μ, satisfying three further identities. An alternative presentation of a monad involves adjoint functors: we consider some of the numerous ways of characterising the existence of a pair of adjoint functors.

Moggi [12] showed that useful notions of computations can be represented by monads, and Wadler, in [14] and subsequent papers, expounded the idea for use in functional programming languages. Examples of monads are the exception monad (to model a computation which may fail), the non-determinism monad (a computation which may produce multiple results).

Distributive Laws for Monads Given two monads, these can sometimes, but not always, be combined to form a monad (for example, describing a computation which may fail, but otherwise may produce multiple results). Jones &

* Supported by an Australian Research Council Large Grant

Duponcheel [6] give a detailed account of various ways to compose monads to form a compound monad. It turns out that one of their constructions is equivalent to the "distributive law" of the category theory literature. Barr & Wells[1, §9.2] give five conditions for a compound monad to be obtained from a distributive law, but we found that two of these are redundant. The approach of Barr & Wells uses a monad in the Eilenberg-Moore category of another monad ; our approach uses a monad in the Kleisli category of another monad.

HOL and HOL$_\omega$ There are potentially different ways to formalise category theory. In a prover such as HOL, there is the limitation that the logic is essentially based on sets. If we overlook this we can model a category as a set of objects, and a set of arrows each of which has a source and target object, and a partial function for composition. Here lies a difficulty: one is forever proving that the target of one arrow is the source of another, before one can compose them. While this particular task is well suited to automation, it is a significant complication. In a dependently typed system, the type of arrows could be parameterised over the source and target objects. In [5], Huet & Saibi describe a formalisation in Coq which addresses this problem essentially by creating a type for the set of arrows between each pair of objects.

In [7], Kozen, Kreitz and Richter describe a formalisation of category theory (in Nuprl), and briefly survey some other formalisations. These explicitly state and prove requirements such as that composing arrows require the target and source to match, and that a functor preserves source and targets of an arrow. Proofs in such formalisations tend to involve a vary large number of steps of this nature, so that success requires a high degree of automation.

Another approach is to use the types of the HOL logic as the objects of a category, as in [4]. This removes the difficulty mentioned above, since the type system can be used to avoid trying to compose arrows whose target and source do not match. However it creates other difficulties, as discussed in more detail by Homeier [4], which result from the fact that the action of a functor on arrows, and of a natural transformation, become polymorphic functions, and the action of a functor on objects (types) is a type constructor.

Homeier [4] developed HOL$_\omega$ on top of HOL to solve these difficulties. Whereas HOL is based on a simply-typed lambda calculus (with first-order type constructor constants and a distinguished boolean type), HOL$_\omega$ also has universal types ($\forall\alpha.\sigma$), and terms may be abstracted over types and applied to types, as in System F. Types are of arbitrary kind, so there are type constructor variables, including higher-order type constructors, and the language of types includes abstraction and application as well as universal quantification. In addition, a boolean term can be universally or existentially quantified over a type. Full details are in [4]. HOL$_\omega$ is compatible with HOL, that is, definitions and proof texts which use only the features of HOL are accepted unchanged in HOL$_\omega$.

In [4] Homeier used HOL$_\omega$ to prove results on categories and monads, using a formalisation where arrows are functions. We found it easy to generalise this, so the type of arrows between objects (types) α and β is not the function type

$\alpha \to \beta$ but, using a variable arrow type constructor A, is $(\alpha, \beta)\ A$. This lets us define dual categories, and, importantly, the Kleisli category of a monad (§3).

In using this system to perform proofs in category theory, therefore, we have made the simplifying assumptions that the objects of a category can be equated to the types of HOL_ω. As we discuss later, this limits the results we can express and prove. It also means that in a logical sense, our results do not apply to all categories, since the logical model underlying HOL is based on sets, but we assert that our proofs go through all the same steps as one would in a pen-and-paper proof, and, as always, the advantage of mechanisation is that none of these steps are overlooked. The value of proceeding in this way is that the proofs can be more succinct, since the type system guarantees certain things which would otherwise have to be checked explicitly. Essentially we have a "light-weight" implementation of categories which enables simple proofs of certain results. We go on to show that this implementation is sufficient to prove new results on a distributive law for monads: our result simplifies the definition of compatibility for monads, and we also give some elegant proofs of existing results.

In §2, we introduce the representation of categories in HOL_ω, and focus on type issues by discussing definitions and theorems about the dual of a category. We then give several equivalent characterisations of pairs of adjoint functors. In §3 we discuss monads, their characterisations, and the Kleisli category of a monad. In §4 we discuss composition of monads, and where this arises from a distributive law, and from a monad in the Kleisli category of another monad.

At present HOL_ω is a branch in the `subversion` HOL repository, and the proofs referred to in the text are contained among the HOL_ω examples, located at `https://hol.svn.sf.net/svnroot/hol/branches/HOL-Omega/examples/HolOmega`, with some files in the subdirectory `interim`. Where results are mentioned in the text, the HOL code of relevant definitions and theorems are given in the Appendix, which is available in `http://users.rsise.anu.edu.au/~jeremy/pubs/holw-cm/`. A name appearing in the text, as (`thm_name`) is the name of a theorem or defined constant; these are shown in the Appendix.

2 Categorical Results ; Adjoint Functors

2.1 Preliminary Results

Our first illustration of using HOL_ω relates to adjoint functors between two categories. Here our work also makes the assumption that the two categories have the same objects (since we use, for objects, HOL types), but we assert this does not affect the proof steps we need to perform.

We assume a basic knowledge of categories, functors and natural transformations. See, eg, [10], [1], [2], [13]. We will write composition of arrows (morphisms) right-to-left, thus $g \circ f$ where the source object of g is the target object of f.

Here we must put in a word about notation. In HOL (as in its implementation and user interface language, Standard ML), type constructors are written postfix, so α `list set` is the type of sets of lists of things of type α. There are no higher-order type constructors, but type constructors can have multiple arguments, so

(α, β) `prod` is the type of pairs with components of type α and β. In HOL_ω this becomes a curried higher-order (type) function, so (α, β) `prod` equates to β (α `prod`). This means that while the action of a functor on arrows is written on the left, the action of a functor on objects (types) is written on the right.

In [4] Homeier proved many results about categories, functors and natural transformations, in the setting where objects are types and arrows are functions between them (in the HOL_ω theory `functor`). However, to be able to deal with the dual of a category we need a more general setting where arrows are of variable type (α, β) A, or `('a,'b) 'A`, for source and target types α and β. (In HOL a type *variable* is preceded by a prime). We proved general results in this setting in theory `category`, and our results about adjoint functors in theory `g_adjoint`.

Since our initial purpose of this generalisation was to allow for dual, or opposite, categories, our first examples of type issues in HOL_ω are in this setting.

`dual_comp` takes an arrow composition function `comp` and returns the appropriate function for composition of the arrows in the reverse direction. Note that it reverses not only the order of the two arrow arguments but also the order of the three type (object) arguments involved. If the types of arrows, originally, are (α, β) A, then, when we are using arrows backwards, the type of an arrow with source α and target β is (α, β) A C, which is defined to be (β, α) A, using the pre-defined (higher-order) type operator C. Likewise, `o_arrow` is defined as a type constructor to express the type of an arrow composition function. Note that `o_arrow` is parameterised over the arrow type constructor A; since, given A, an arrow composition function must apply to *all* objects, it is universally quantified over the three types (objects) involved. We defined a predicate `category` (see Appendix A.1), asserting that the composition function $\circ$ is associative, and that the identity arrow for each type (object) is the identity of $\circ$. We show part of it below: the arguments are the arrow type constructor A, the identity arrow function, of type A *id*, which is, for each object (type) α, an arrow of type $(\alpha, \alpha)A$ and the arrow composition function, of type A *o_arrow*. Then we could prove (`category_dual`) that the dual of a category is a category.

$$(\alpha, \beta)\ A\ C = (\beta, \alpha)\ A \qquad\qquad A\ id = \forall \alpha.\ (\alpha, \alpha)A$$
$$A\ o\_arrow = \forall \alpha\beta\gamma.\ (\beta, \gamma)A \to (\alpha, \beta)A \to (\alpha, \gamma)A$$
$$category\ [:\ A\ :]\ (id : A\ id, comp : A\ o\_arrow) = \ldots$$
$$dual\_comp\ comp\ [:\ \gamma, \beta, \alpha\ :]\ f\ g = comp\ [:\ \alpha, \beta, \gamma\ :]\ g\ f$$

Getting the statement of the theorem `category_dual` to typecheck was difficult. Initally we had defined the `category` predicate without the explicit type argument `'A`. In earlier examples this had caused no problems. Although in this system type-checking involves higher-order unification of type expressions, which doesn't give most general unifiers and therefore principal types, the algorithm would manage when the types of arrows consistently used the same type constructor. In this case we found that the only solution seemed to be a type parameter (as putting more type annotations in did not succeed). This required us to redefine the predicate `category` to have a type parameter. This would normally require every use of it to be changed to include the type argument, but in fact since the inclusion of type arguments is optional in HOL$_\omega$, changing the definition to include type arguments did not require changing the other previous uses of `category` in the sources.

Recall that a functor (between categories with arrow type constructors C and D, say) comprises a mapping of objects (in our HOL$_\omega$ model this is a type constructor, say F), and a mapping of arrows. The arrow map must satisfy

– preservation of sources and targets of objects: the type of the arrow map, $\forall\alpha\beta.\ (\alpha, \beta)C \to (\alpha\ F, \beta\ F)D$ (*type abbreviation* `g_functor`) ensures this
– preservation of identity and composition of arrows: the *predicate* `g_functor` (Appendix A.1) specifies this

Stating duality theorems provided unexpected difficulties. Given the definition of a functor, ie, so far as types are concerned, a function on objects (a type constructor) and a function on arrows, this should be unchanged for dual categories. This is not so! The type of the arrow map is quantified over the source and target types, so dualising the functor requires changing the order of these two quantifications. Function `g_dual_functor` (Appendix A.1) dualises a functor in this way. Once the statement was formalised correctly, it was trivial to prove (as `g_functor_dual`) that the "dual" of a functor is also a functor.

2.2 Adjoint Functors

A pair of adjoint functors can be defined in several ways. We first give two equivalent common definitions.

Definition 1. *For categories $\mathcal{C}$ and $\mathcal{D}$, functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$, form an* adjunction *(written $F \dashv G$) if*

(i) there is a natural transformation $\eta : I_\mathcal{C} \to G \circ F$, and for each arrow $f : A \to GB$ in $\mathcal{C}$, there is a unique arrow $g : FA \to B$ in $\mathcal{D}$ (call it $g = f^\sharp$) such that $Gg \circ \eta = f$, or
(ii) there is a natural transformation $\epsilon : F \circ G \to I_\mathcal{D}$ and for each arrow $g : FA \to B$ in $\mathcal{D}$, there is a unique arrow $f : A \to GB$ in $\mathcal{C}$ (call it $f = g^\flat$) such that $\epsilon \circ Ff = g$

These definitions are dual and equivalent, and the functions $\cdot^\sharp$ and $\cdot^\flat$ are mutually inverse. Then these give $\epsilon = I_\mathcal{C}^\sharp$ and $\eta = I_\mathcal{D}^\flat$.

So our first significant exercise in the use of HOL$_\omega$ was to prove the equivalence of these two definitions. Since they are dual, we found an intermediate formulation which is self-dual, which is Theorem 2(iii), so having proved that (i) and (iii) are equivalent, we could use duality to prove that (ii) and (iii) are equivalent. This was actually not as easy as it sounds — we had to formulate, prove and use many lemmas expressing the duality between each component of (ii) and the corresponding component of (i). Cutting, modifying and pasting proofs might have been easier!

Performing these proofs had a surprising outcome: having formulated the definitions above and done the proofs, we found that not all the assumptions were necessary. While adjoint pairs are usually defined assuming F and G are functors, we found that, for F in Definition 1(i), and for G in Definition 1(ii), it is not necessary. This is found in Cockett [2, Proposition 2.2.2] (though he says it was known earlier). He then provided, essentially, Theorem 2(iv), a criterion for an adjoint pair not assuming either F or G to satisfy the functor equalities. He then discovered a similar criterion and discussion in [8, Definitions 109,110], and the result that the two parts of Theorem 2(iv) are equivalent, in [3, §3.1].

In the following theorem, $F, G, \eta, \epsilon, \cdot^\sharp$ and $\cdot^\flat$ are assumed to be of the appropriate "types" without necessarily satisfying the equalities for a functor, natural transformation, etc. But it *is* assumed that they return arrows in the correct categories, with the correct source and target objects. Note how the last few conditions in each criterion are definitional: so, for example, when starting with (i), with G, η and $\cdot^\sharp$, you can define F, ϵ and $\cdot^\flat$ to make the last three conditions true. The HOL_ω theorems underlying this result are described in Appendix A.2.

Theorem 2. *Let $\mathcal{C}$ and $\mathcal{D}$ be categories. Then the following are equivalent. (Where relevant, conditions are for* all *f, g and h).*

(i) *G is a functor, η is natural, $Gg \circ \eta = f$ iff $g = f^\sharp$, $\epsilon = I_{\mathcal{C}}^\sharp$, $g^\flat = Gg \circ \eta$ and $Ff = (\eta \circ f)^\sharp$*

(ii) *F is a functor, ϵ is natural, $\epsilon \circ Ff = g$ iff $f = g^\flat$, $\eta = I_{\mathcal{D}}^\flat$, $f^\sharp = \epsilon \circ Ff$ and $Gg = (g \circ \epsilon)^\flat$*

(iii) *F and G are functors, $Gg \circ \eta = f$ iff $\epsilon \circ Ff = g$, $g^\flat = Gg \circ \eta$ and $f^\sharp = \epsilon \circ Ff$*

(iv) *$\cdot^\sharp$ and $\cdot^\flat$ are mutually inverse, $(f \circ h)^\sharp = f^\sharp \circ (\eta \circ h)^\sharp$ and/or $(h \circ g)^\flat = (h \circ \epsilon)^\flat \circ g^\flat$ (which are equivalent), $\eta = I_{\mathcal{D}}^\flat$, $\epsilon = I_{\mathcal{C}}^\sharp$, $Ff = (\eta \circ f)^\sharp$ and $Gg = (g \circ \epsilon)^\flat$*

(v) *[13, Proposition 7.6] F and G are functors, η and ϵ are natural, $G\epsilon \circ \eta G = I_{\mathcal{C}}G$, $\epsilon F \circ F\eta = I_{\mathcal{D}}F$, $g^\flat = Gg \circ \eta$ and $f^\sharp = \epsilon \circ Ff$*

3 Monads

Since the language of HOL is close to that of functional programming languages we will present monads in a way that is closer to Wadler's treatment than that of category theory. Thus a monad involves functions of types (read $\rightsquigarrow$ as $\rightarrow$):

$$\begin{aligned}
\mathit{unit} &: \alpha \rightsquigarrow \alpha M \\
\mathit{map} &: (\alpha \rightsquigarrow \beta) \rightarrow (\alpha M \rightsquigarrow \beta M) \\
\mathit{join} &: \alpha M M \rightsquigarrow \alpha M \\
\mathit{ext} &: (\alpha \rightsquigarrow \beta M) \rightarrow (\alpha M \rightsquigarrow \beta M) \\
\odot &: (\beta \rightsquigarrow \gamma M) \rightarrow (\alpha \rightsquigarrow \beta M) \rightarrow (\alpha \rightsquigarrow \gamma M) && \text{(infix)}
\end{aligned}$$

To relate this to category theory, the type constructor M is the action of a functor on objects, and *map* is the action of a functor on arrows. The polymorphic functions *unit* and *join* are the natural transformations η and μ; the other functions will be mentioned later.

We note here that in the case of the types of *map*, *ext* and $\odot$, when we generalise from the category of functions to a category of general arrow type, the '$\leadsto$' symbols represent this general arrow type, whereas the '$\to$' symbols continue to represent functions. This is why we prefer to use *ext* rather than *bind*, as in Wadler [14], and in [4], in stating the monad rules: $m\ bind\ f = ext\ f\ m$.

If a monad is defined in terms of *unit*, *map* and *join*, rules (1) to (7) must be satisfied. This is the "monoid form" presentation of an algebraic theory of Manes [11, Chapter 1, Definition 3.17]. (`g_umj_monad_exp`)

$$map\ id = id \tag{1}$$
$$map\ f \circ map\ g = map\ (f \circ g) \tag{2}$$
$$unit \circ f = map\ f \circ unit \tag{3}$$
$$join \circ map\ (map\ f) = map\ f \circ join \tag{4}$$
$$join \circ unit = id \qquad \mu \circ \eta M = idM \tag{5}$$
$$join \circ map\ unit = id \qquad \mu \circ M\eta = idM \tag{6}$$
$$join \circ map\ join = join \circ join \qquad \mu \circ M\mu = \mu \circ \mu M \tag{7}$$

$$ext\ f = join \circ map\ f \tag{8}$$

Here rules (1) and (2) express that *map* is the action on arrows of a functor, (3) and (4) express that η and μ are natural transformations, and (5) to (7) are the three identities given in Manes [11, Chapter 1, Definition 3.17]. We also give these latter in common category theory notation. Note the use of μM and ηM, composition of a natural transformation with a functor: while this is not reflected in the form of the axioms as found in [14], it can be seen in the type annotations of the respective terms in HOL_ω, as shown. Note that most of the type annotations have been edited out!

```
comp (join [:'a:]) (unit [:'a 'M:]) = id [:'a 'M:]
comp (join [:'a:]) (map (unit [:'a:])) = id [:'a 'M:]
comp (join [:'a:]) (map (join [:'a:])) =
                     comp (join [:'a:]) (join [:'a 'M:])
```

Then (8) defines *ext* in terms of *join* and *map*.

Alternatively, a monad can be defined in terms of *unit* and *ext*. Wadler [14] shows the equivalence to analogues (in terms of *bind*) of rules (E1), (E2) and (E3), which are given by Moggi [12]. This is the "extension form" presentation of an algebraic theory of Manes [11, Chapter 1, §3, Exercise 12]. (`Komonad_thm`)

$$ext\ f \circ unit = f \tag{E1}$$
$$ext\ unit = id \tag{E2}$$
$$ext\ (g \odot f) = ext\ g \circ ext\ f \tag{E3}$$

$$join = ext\ id \tag{E4}$$
$$map\ f = ext\ (unit \circ f) \tag{E5}$$
$$g \odot f = ext\ g \circ f \tag{E6}$$
$$ext\ g = g \odot id \tag{E6$'$}$$

If a monad is defined by giving *unit*, *map* and *join*, satisfying the rules (1) to (7), and if *ext* and $\odot$ are defined by (8) and (E6), then rules (E1) to (E5) hold. Conversely, if a monad is defined by giving *unit*, *ext* and $\odot$, satisfying rules (E1), (E2) and (E3) then (E6) holds, and if *join* and *map* are defined by (E4) and (E5), then the rules (1) to (8) hold. We can express this as a equivalence:

Theorem 3 (`g_umj_iff_Kdomonad`). *(1) to (8) and (E6)* $\Leftrightarrow$ *(E1) to (E5)*

The HOL_ω proof of Theorem 3 was an application of a theorem involving compound monads, see Appendix A.11 for details.

Now from (E1) we get $\mathit{ext}\, f = \mathit{ext}\, g \Rightarrow f = g$, whence (E2) and (E3) give that $\odot$ is associative, and has the identity *unit*. Thus there is a category, whose objects are types, whose arrows from α to β are arrows (functions) of "type" $\alpha \rightsquigarrow \beta M$, and whose identity and composition are unit_M and $\odot$ (`Komonad_IMP_Kcat`). This is the *Kleisli* category, $\mathcal{C}_M$, of the monad M. We use $\alpha \rightsquigarrow_M \beta = \alpha \rightsquigarrow \beta M$ to represent the "types" of its arrows. Then, by (E2) and (E3), *ext* is the action on arrows of a functor from the Kleisli category $\mathcal{C}_M$ into the original category $\mathcal{C}$ (`Komonad_IMP_ext_f`); its action on objects is just the type constructor M.

The use of a general arrow type in our HOL_ω development enables us to express results about the Kleisli category.

We define the Kleisli arrow so that $(A, M, \alpha, \beta)\ \mathit{Kleisli} = \alpha \rightsquigarrow_M \beta$. Recall that this is syntactic sugar for a curried type operator, $\beta(\alpha(M(A\ \mathit{Kleisli})))$. We partially apply it, so $M(A\ \mathit{Kleisli})$ is the arrow type operator in the Kleisli category $\mathcal{C}_M$, and the theorem `Komonad_IMP_Kcat` was entered as shown below, and the type of the arrow composition in $\mathcal{C}_M$ is then `('A, 'M) Kleisli o_arrow`.

```
Komonad_IMP_Kcat : |- category [:'A:] (id, comp) ==>
    Komonad (id, comp) (unit, ext, kcomp) ==>
    category [: ('A, 'M) Kleisli :] (unit, kcomp)
```

The relationship between adjoint functors and monads is well known: every pair of adjoint functors F and G gives a monad, whose action on objects is $G \circ F$, whose *unit* (η) is just η (!), and where $\mathit{ext}\, f$ is $G(f^\sharp)$ (`g_adjf1_IMP_Kmonad`). In doing the HOL_ω proofs we noticed that this result does not even require that η be a natural transformation. Every monad is obtainable from a pair of adjoint functors, via (inter alia) the Kleisli category, the adjoint functors being *ext* and the functor which is the identity on objects, and takes arrow f to $\mathit{unit} \circ f$.

We proved some of these results in HOL_ω, as detailed in Appendix A.3. We record a lemma which we find particularly useful later.

Lemma 4. *In a monad the following are equivalent*

(i) $\mathit{ext}\ g = g \circ \mathit{join}$
(ii) $g = \mathit{ext}\ (g \circ \mathit{unit})$
(iii) there exists f *such that* $g = \mathit{ext}\, f$
(iv) for all h, $\mathit{ext}\ (g \circ h) = g \circ \mathit{ext}\ h$

4 Compound Monads

If M and N, with associated functions, are monads, we can sometimes, but not always, combine these two monads to form a third.

Compound monads can arise naturally and be practically useful ([14], [6]). In this section we consider a construction for compound monads and discuss the conditions under which it is possible. When further conditions are satisfied, then we have the distributive law for monads described by Manes [11] and Barr & Wells [1]: see §4.2. In this case the monads are "compatible" [1, §9.2], and our results show how some of the conditions given in [1] for this are redundant.

4.1 Compound Monads *via* Partial Extension

Let M be a monad, and consider compound monads where the compound monad type is αNM, that is $(\alpha N)M$. We will call it the monad MN, in line with the usual notation for composition of functors. To define a compound monad MN, we will need a function ext_{MN}, which "extends" a function f from a "smaller" domain, α, to a "larger" one, αNM. Consider, therefore, a "partial extension" function *pext* which does part of this job:

$$\begin{aligned} ext_{MN} &: (\alpha \rightsquigarrow \beta NM) \to (\alpha NM \rightsquigarrow \beta NM) \\ pext &: (\alpha \rightsquigarrow \beta NM) \to (\alpha N \rightsquigarrow \beta NM) \\ pext &: (\alpha \rightsquigarrow_M \beta N) \to (\alpha N \rightsquigarrow_M \beta N) \end{aligned}$$

We will prove that rules (E1K) to (E3K) are sufficient to define a compound monad using such a function *pext*. We assume nothing about the functions $\odot_{MN}$ or $unit_{MN}$, except that they have the appropriate types, or that N is a monad.

$$pext\ f\ \odot_M\ unit_{MN} = f \tag{E1K}$$
$$pext\ unit_{MN} = unit_M \tag{E2K}$$
$$pext\ (g\ \odot_{MN}\ f) = pext\ g\ \odot_M\ pext\ f \tag{E3K}$$

$$kjoin = pext\ unit_M \tag{E4K}$$
$$kmap\ f = pext\ (unit_{MN}\ \odot_M\ f) \tag{E5K}$$

$$g\ \odot_{MN}\ f = pext\ g\ \odot_M\ f \tag{E6K}$$
$$pext\ g = g\ \odot_{MN}\ unit_M \tag{E6$'$K}$$

Observing how the second type expression for *pext* above looks like that for *ext* in §3, and by comparing (E1K) to (E3K) with (E1) to (E3) we see that we have the three rules needed for a monad N in $\mathcal{C}_M$, the Kleisli category of M. We will refer to this monad as N_M. Its *unit* is $unit_{MN}$; recall that the identity and composition in the category are $unit_M$ and $\odot$.

$$\begin{aligned} unit_M &: \alpha \rightsquigarrow_M \alpha \\ unit_{MN} &: \alpha \rightsquigarrow_M \alpha N \\ \odot_M &: (\beta \rightsquigarrow_M \gamma) \to (\alpha \rightsquigarrow_M \beta) \to (\alpha \rightsquigarrow_M \gamma) \end{aligned}$$

Thus the treatment of a single monad described in §3 applies to this monad. We define the counterparts of *map* and *join*, calling them *kmap* and *kjoin*: note how rules (E4K) and (E5K) correspond to (E4) and (E5). As in §3, we deduce (E6K), giving $\odot_{MN}$ in tems of *pext*. We will refer to rule (r) for this monad as (rK); likewise we may refer to rules (rM), (rN) or (rMN) where necessary.

We also obtain rules (1K) to (8K) which are the counterparts of (1) to (8), and these hold by Theorem 3 for this monad N_M. These are set out in Appendix A.4. Likewise Theorem 4 holds for N_M. This monad N_M in $\mathcal{C}_M$, the Kleisli category for M, gives rise to a further Kleisli category, which we may describe as the Kleisli category for N_M in $\mathcal{C}_M$. Its identity is $unit_{MN}$ and its composition function is $\odot_{MN}$. It turns out that this is also the Kleisli category of a monad MN in $\mathcal{C}$.

Theorem 5 (`Ko_pext_cm`). *Assume that M is a monad and that functions pext, $\odot_{MN}$ and $unit_{MN}$ of the appropriate types are given, satisfying rules (E1K) to (E3K). Then, using (EC) to define ext_{MN}, $unit_{MN}$, ext_{MN} and $\odot_{MN}$ define a monad, and (PE) and (J1S) hold.*

$$ext_{MN}\ f = ext_M\ (pext\ f) \tag{EC}$$

$$pext\ f = ext_{MN}\ f \circ unit_M \tag{PE}$$

$$ext_M\ (ext_{MN}\ f) = ext_{MN}\ f \circ join_M \tag{J1S}$$

Proof. By rules (E1K) to (E3K), *pext*, $\odot_{MN}$ and $unit_{MN}$ give a monad in $\mathcal{C}_M$.

(PE) follows from (EC) and (E1M).

By Theorem 4 for M, (iii) $\Rightarrow$ (i), (J1S) follows from (EC).

It is now trivial to use (EC) to prove rules (E1MN) to (E3MN). Rules (E2MN) and (E3MN) each follows directly from (EC) and the same rule for M and for N_M. Rule (E1MN) is proved by (EC), (E6M) and (E1K). □

We may next ask which compound monads can be constructed from such a function *pext*, satisfying rules (E1K) to (E3K). The previous theorem provides a necessary condition, namely that (J1S) holds. In fact, this condition is also sufficient. The key step in the proof of this is to define *pext* from ext_{MN} by applying Theorem 4 for M, (i) $\Rightarrow$ (ii), to (J1S). We further note that the special case $f = id$ of (J1S) (called (J1), shown below) implies it in general.

Theorem 6 (`cm_if_J1o,J1_IMP_ext_pext`). *Let M and MN be monads, such that (J1S) (see Theorem 5) holds. Then $\odot_{MN}$ and $unit_{MN}$ also define a monad in the category $\mathcal{C}_M$, and, using (PE) to define pext, (EC) holds.*

This shows that, given a monad M, that the compound monads MN obtainable using the construction via a monad N in $\mathcal{C}_M$ are precisely the monads MN such that (J1S) (or (J1)) is satisfied. See theorem `cm_Ko_J1S` in Appendix A.4.

4.2 Compound Monads via a Distributive Law

Jones & Duponcheel [6] consider when a compound monad may be obtained using various constructions. They consider only compound monads which satisfy

(UC) and (MC), and we will assume these without further comment. In §3.2 they showed a construction using a function *prod*, satisfying certain properties P(1) to P(4), which can be related to (E1K) to (E3K). In §4.1 they showed that this construction is applicable if and only if their condition J(1) (which is our (J1)) holds. In fact these are related: $\mathit{prod} = \mathit{pext}\ \mathit{id}$ and $\mathit{pext}\ f = \mathit{prod} \circ \mathit{map}_N\ f$.

Similarly in §3.3 and §4.2 they showed a construction using a function *dorp*, satisfying certain properties (their D(1) to D(4)), which is applicable if and only if their condition J(2) (our (J2)) holds. Then in §3.4 and §4.3 they showed a construction using a function *swap*, satisfying certain properties S(1) to S(4), which is applicable if and only if J(1) and J(2) hold.

$$\mathit{unit}_{MN} = \mathit{unit}_M \circ \mathit{unit}_N \tag{UC}$$
$$\mathit{map}_{MN}\ f = \mathit{map}_M\ (\mathit{map}_N\ f) \tag{MC}$$
$$\mathit{ext}_M\ \mathit{join}_{MN} = \mathit{join}_{MN} \circ \mathit{join}_M \tag{J1}$$
$$\mathit{ext}_{MN}\ (\mathit{map}_M\ \mathit{join}_N) = \mathit{map}_M\ \mathit{join}_N \circ \mathit{join}_{MN} \tag{J2}$$

They observe that both J(1) and J(2) are of a certain form, which is that of Theorem 4(i). Therefore their conditions are, in effect, that join_{MN} is of the form $\mathit{ext}_M\ f$, and that $\mathit{map}_M\ \mathit{join}_N$ is of the form $\mathit{ext}_{MN}\ g$.

Barr & Wells [1, §9.2] describe a construction of compound monads using a "distributive law" λ. A distributive law for monads M and N is a natural transformation λ satisfying four rules shown below. In fact λ is the *swap* of [6].

$$\begin{aligned} \mathit{swap} \circ \mathit{map}_N\ \mathit{unit}_M &= \mathit{unit}_M \\ \lambda \circ N\eta_M &= \eta_M N \end{aligned} \tag{D1}$$
$$\begin{aligned} \mathit{swap} \circ \mathit{unit}_N &= \mathit{map}_M\ \mathit{unit}_N \\ \lambda \circ \eta_N M &= M\eta_N \end{aligned} \tag{D2}$$
$$\begin{aligned} \mathit{swap} \circ \mathit{map}_N\ \mathit{join}_M &= \mathit{ext}_M\ \mathit{swap} \circ \mathit{swap} \\ \lambda \circ N\mu_M &= \mu_M N \circ M\lambda \circ \lambda M \end{aligned} \tag{D3}$$
$$\begin{aligned} \mathit{swap} \circ \mathit{join}_N &= \mathit{map}_M\ \mathit{join}_N \circ \mathit{swap} \circ \mathit{map}_N\ \mathit{swap} \\ \lambda \circ \mu_N M &= M\mu_N \circ \lambda N \circ N\lambda \end{aligned} \tag{D4}$$

Conversely, they define five conditions, (C1) to (C5), for monads M, N and MN to be "compatible". (C1) is (UC), and their treatment presupposes (MC). In fact (C4) and (C5) are precisely (J1) and (J2). They show that MN is compatible with M and N if and only if it can be constructed using a distributive law.

$$\begin{aligned} \mathit{ext}_{MN}\ \mathit{unit}_M &= \mathit{map}_M\ \mathit{join}_N \\ \mu \circ MN\eta_M N &= M\mu_N \end{aligned} \tag{C2}$$
$$\begin{aligned} \mathit{join}_{MN} \circ \mathit{map}_M\ \mathit{unit}_N &= \mathit{join}_M \\ \mu \circ M\eta_N MN &= \mu_M N \end{aligned} \tag{C3}$$
$$\begin{aligned} \mathit{ext}_M\ \mathit{join}_{MN} &= \mathit{join}_{MN} \circ \mathit{join}_M \\ \mu_M N \circ M\mu &= \mu \circ \mu_M NMN \end{aligned} \tag{C4}$$
$$\begin{aligned} \mathit{map}_M\ \mathit{join}_N \circ \mathit{join}_{MN} &= \mathit{ext}_{MN}\ (\mathit{map}_M\ \mathit{join}_N) \\ M\mu_N \circ \mu N &= \mu \circ MNM\mu_N \end{aligned} \tag{C5}$$

In fact the definition of compatibility can be simplified, since two of the five conditions given by Barr & Wells in [1, §9.2] are redundant. See Appendix A.5 for the HOL$_\omega$ theorems.

Theorem 7. *(C2) ⇔ (C5) and (C3) ⇔ (C4)*

Proof. (C2) ⇒ (C5): this is just Theorem 4 for MN, (iii) ⇒ (i)

(C5) ⇒ (C2): by Theorem 4, (i) ⇒ (ii), $map_M\ join_N = ext_{MN}\ f$ where $f = map_M\ join_N \circ unit_{MN} = unit_M \circ join_N \circ unit_N = unit_M$, by (UC), (3M) and (5N)

(C4) ⇒ (C3): by Theorem 4 for M, (i) ⇒ (ii), we get

$$\begin{aligned} join_{MN} \circ map_M\ unit_N &= ext_M\ (join_{MN} \circ unit_M) \circ map_M\ unit_N \\ &= ext_M\ (join_{MN} \circ unit_M \circ unit_N) && \text{(8M,2M)} \\ &= ext_M\ id = join_M && \text{(UC,5MN)} \end{aligned}$$

(C3) ⇒ (C4): from (C3) we prove the stronger (C3S) using (8MN), (8M), (MC), (2M) and (3N). Substituting the right-hand sides of this and (C3) into the statement of (C4) gives (*), which holds by (7MN).

$$ext_{MN}\ f \circ map_M\ unit_N = ext_M\ f \qquad \text{(C3S)}$$
$$ext_{MN}\ join_{MN} \circ map_M\ unit_N = join_{MN} \circ join_{MN} \circ map_M\ unit_N \qquad (*)$$

There are some intriguing aspects to Theorem 7. As Michael Barr pointed out, there is a duality between (C2,C5) and (C3,C4), when these are written in category theory notation (for $X_1 \circ X_2 \circ \ldots$, you reverse the order of the symbols *within* each X_i, and interchange M and N). In the same way, (D2) and (D4) are dual to (D1) and (D3). Further, it seems to be possible to dualise the relevant proofs. Often the reasons for proof steps change. For example, $M(f \circ g) = Mf \circ Mg$ when M is a functor, whereas $(f \circ g)M = fM \circ gM$ (for natural transformations f, g) by definition. And when the proofs are written out in this notation (see Appendix A.6) a step depending on the naturality of μ becomes a step depending on the naturality of η_N.

These observations by Michael Barr enabled us to find the proof of (C3) ⇒ (C4). At first we had conjectured (C3) ⇒ (C4) but were unable to prove it. On the other hand its dual, the proof (C2) ⇒ (C5), was an instance of the rather trivial Theorem 4. So to find the proof of (C3) ⇒ (C4), we translated our simple proof of (C2) ⇒ (C5) into the steps shown in Appendix A.6, and dualised them.

Note further that although (C4) and (C5) have this duality as they stand; however their similarity that they are both of the form of Theorem 4(i) applies to the symmetric equivalent of (C5). Another curious point is that the identity $ext_{MN}\ unit_M \circ map_M\ unit_N = id$ follows from *either* (C2) *or* (C3).

We have now shown that the compatibility conditions (C2) to (C5) are just J(1) and J(2) of Jones & Duponcheel [6, §4]. Since their S(1) is just the requirement that *swap* is a natural transformation, it follows that their conditions S(2) to S(4) on *swap* must be equivalent to (D1) to (D4). In fact S(2) and S(3) are just (D2) and (D1) respectively. For S(4), define *prod* and *dorp* from *swap*, then

S(4) is as shown. We indicate in Appendix A.7 how to prove S(4) directly from (D3) and (D4), and vice versa.

$$\mathit{prod} = \mathit{map}_M\ \mathit{join}_N \circ \mathit{swap} \qquad \mathit{dorp} = \mathit{ext}_M\ \mathit{swap}$$
$$\mathit{prod} \circ \mathit{map}_N\ \mathit{dorp} = \mathit{dorp} \circ \mathit{prod} \tag{S(4)}$$

We now indicate how we used the properties of the monad N_M to obtain easy proofs that a compound monad satisfying the compatibility conditions provides a distributive law. By Theorem 6, compatibility conditions (C3/C4), ie (J1), imply that the monad NM is derived from a monad N_M in the Kleisli category $\mathcal{C}_M$. This gives us functions *pext*, *kjoin* and *kmap*.

Lemma 8. *Assume MN is constructed as in §4.1. Then*

$$\mathit{pext}\ (g \circ f) = \mathit{pext}\ g \circ \mathit{map}_N\ f \tag{PO}$$
$$\mathit{kmap}\ (g \circ f) = \mathit{kmap}\ g \circ \mathit{map}_N\ f \tag{KO}$$

Proof. (PO): Uses (PE), (8MN), (2MN), (MC), (3M) and (PE) again.
(KO): Uses (E5K), (E6M), (PO) and (E6M) and (E5K) again. □

If we assume also the compatibility condition (C2), ie (J2), then, using (PE), (C2) and (3M), we can show $\mathit{kjoin} = \mathit{unit}_M \circ \mathit{join}_N$. We define the distributive law λ, ie, *swap*, as in [1, §9.2] (and in [6, §4.3]), namely $\mathit{swap} = \mathit{pext}\ (\mathit{map}_M\ \mathit{unit}_N)$, which is equal to *kmap id* (see Appendix A.9 for these). With these results we can translate the conditions (D1) to (D4), and S(4) of Jones & Duponcheel [6], into the following, which are all immediate results about the monad N_M in the Kleisli category $\mathcal{C}_M$.

$$\mathit{kmap}\ \mathit{unit}_M = \mathit{unit}_M \tag{D1$'$}$$
$$\mathit{kmap}\ \mathit{id} \odot_M \mathit{unit}_{MN} = \mathit{unit}_{MN} \odot_M \mathit{id} \tag{D2$'$}$$
$$\mathit{kmap}\ (\mathit{id} \odot_M \mathit{id}) = \mathit{kmap}\ \mathit{id} \odot_M \mathit{kmap}\ \mathit{id} \tag{D3$'$}$$
$$\mathit{kmap}\ \mathit{id} \odot_M \mathit{kjoin} = \mathit{kjoin} \odot_M \mathit{kmap}\ (\mathit{kmap}\ \mathit{id}) \tag{D4$'$}$$
$$\mathit{pext}\ (\mathit{kmap}\ \mathit{id} \odot_M \mathit{id}) = \mathit{kmap}\ \mathit{id} \odot_M \mathit{pext}\ \mathit{id} \tag{S4$'$}$$

(D1$'$) to (D4$'$) are just instances of (1K), (3K), (2K) and (4K), and (S4$'$) follows from Theorem 4 for N_M, (iii) ⇒ (iv). That *swap* is a natural transformation is more easily proved using its definition from M and N and natural transformations join_{MN}, unit_M and unit_N, but a proof using the conditions for the monad N_M in $\mathcal{C}_M$ is in Appendix A.8.

To prove the converse, that if the conditions (D1) to (D4) hold, then we can construct a compound monad MN which satisfies the compatibility conditions, is not quite so easy, but for this proof also we have found the easiest way is again to use the conditions for a monad N_M in $\mathcal{C}_M$. We prove the seven rules (1K) to (7K). Details are in Appendix A.10.

5 Conclusions ; Further Work

Ease and Difficulty of using $\mathbf{HOL_\omega}$ Quoting [4, §1], the logic of HOL and its type system has found a "sweet spot", combining expressivity with ease of use. Arguably, the key to this is that its type system is decidable, where typechecking involving first-order unification, which produces principal types for each subterm.

Typechecking in HOL_ω does not have these advantages, and so, inevitably, it can fail. As is usual in HOL (as with programming in Standard ML), a type error is often detected far away from where the programmer's error occurred. In HOL_ω this also occurs, where the "error" is insufficient type annotation.

One solution is to include more type annotations. It is of course easy to get them wrong: if the computer can work them out, it generally gets them right, which is a considerable help for the user.

There are two sorts of type annotations: giving the type of a term (as in HOL, and Standard ML), and giving an explicit type argument to a function, where the function's type requires it. These explicit type arguments seem to be particularly helpful to the type checker, and several times we redefined functions to use them, after finding difficulties in getting a term to typecheck. In particular, we suggest that type arguments rather than free type variables be used in a function definition in the case of higher-order types (type constructors).

In entering a term, such type arguments may be omitted: this is a marvellous feature, since inserting them all would be a considerable burden. In particular, when a function is later redefined to have a type argument, it is a great relief that one does not need to change every use of that function. It should be acknowledged here that HOL_ω is backward compatible with HOL, so that HOL code not using the new features will run unchanged. This is immensely valuable, well worth the extra development effort that was doubtless required.

Value and Capabilities of $\mathbf{HOL_\omega}$ It is clearly possible to prove useful results in HOL_ω. While proving routine results about adjoint functors, we noticed, unexpectedly, that at one point one of the usual assumptions had in fact not been used. This gave us a result which was known, but not to us, and this led to our discussion of minimal presentations of a pair of adjoint functors.

Then we have been able to express and prove results simplifying the conditions given by Barr & Wells for compatibility of two monads M and N, showing that two of their conditions are redundant, and make use in the proofs of the fact that we have one monad in the Kleisli category of another monad. Applying results about a monad directly to a monad in the Kleisli category used both these new features of HOL_ω: explicit type parameters, and higher-order types.

However our formalisation of categories has been based on representing objects as types in the HOL_ω logic. So far as we can see this does not provide a way of expressing results about, for example, the Eilenberg-Moore category of a monad, whose objects are constructed from objects *and* arrows of the original category, where these satisfy certain conditions. This suggests attempting to model categories in a framework with dependent types, such as Coq or Lego.

Our treatment of categories also has some formal limitations. Since HOL$_\omega$ is a set-based logic we have clearly not formally proved results about categories whose collections of arrows are not sets. And since we used HOL$_\omega$ types as objects, our adjoint functors were between categories with the same objects. On the other hand, we have not taken advantage of these facts; all our proof steps are the same as in a proof which formally applies to categories in general.

In considering distributive laws, we made use of the fact that there is a monad, related to N, in the Kleisli category of M. Barr & Wells show that there is a monad, a "lifting" of M, in the Eilenberg-Moore category of N. Since there a duality between the Kleisli and Eilenberg-Moore constructions we suspect some connexion between these results, which we have not yet identified.

Acknowledgements I wish to thank Robin Cockett for helpful discussions about minimal characterisations of pairs of adjoint functors, and Michael Barr for discussions about distributive laws, including the crucial observation about the duality between various compatibility conditions. I wish to thank Peter Homeier and Michael Norrish for their help with the use of HOL$_\omega$ and HOL.

References

1. Michael Barr and Charles Wells. Toposes, Triples and Theories. Springer-Verlag, 1983, or see `http://www.cwru.edu/artsci/math/wells/pub/ttt.html`
2. Robin Cockett. Lecture Notes. `http://pages.cpsc.ucalgary.ca/~robin/class/617/notes.pdf`
3. Kosta Doen. Deductive Completeness. Bull. Symbolic Logic Volume 2, Number 3 (1996), 243–283. `http://www.math.ucla.edu/~asl/bsl/0203/0203-001.ps`
4. Peter Homeier. The HOL-Omega Logic. In: Proc. TPHOLs 2009, LNCS, vol. 5674, pp. 244–259. Springer, Heidelberg (2006)
5. Gérard P. Huet & Amokrane Saïbi. Constructive category theory. In: Plotkin, Stirling, Tofte (eds) Proof, Language, and Interaction, MIT Press, 2000, 239-276.
6. Mark P Jones and Luc Duponcheel. Composing Monads, Research Report YALEU/DCS/RR-1004, Yale University, December 1993
7. Dexter Kozen, Christoph Kreitz, and Eva Richter. Automating proofs in category theory. In: Proc. IJCAR'06, LNAI, vol. 4130, pp 392-407. Springer, August 2006.
8. Paul Levy. Call-By-Push-Value. PhD thesis, Queen Mary and Westfield College, London, 2001. `http://www.cs.bham.ac.uk/~pbl/papers/thesisqmwphd.pdf`
9. Sheng Liang, Paul Hudak, and Mark P Jones. Monad Transformers and Modular Interpreters. In Proc. Principles of Programming Languages (POPL'95), 333–343.
10. Saunders MacLane. Categories for the Working Mathematician. Springer, 1971.
11. Ernest Manes. Algebraic theories. Graduate Texts in Mathematics, Springer, 1976.
12. Eugenio Moggi. Computational lambda-calculus and monads. In Proceedings of Logic in Computer Science (LICS'89), 1989.
13. Daniele Turi. Category Theory Lecture Notes. `http://www.dcs.ed.ac.uk/home/dt/CT/categories.pdf`.
14. Philip Wadler. The Essence of Functional Programming. In Symposium on Principles of Programming Languages (POPL'92), 1992, 1–14.

A HOL$_\omega$ definitions and theorems

A.1 Basic Definitions and Theorems about Categories

```
(in src/bool/boolScript.sml)
val _ = type_abbrev ("C", Type `: \('f:'l => 'k => 'm) 'a 'b. 'a ('b 'f)`);

val _ = type_abbrev ("o_arrow", Type `: \'A.
  (!'a 'b 'c. (('b,'c) 'A) -> (('a,'b) 'A) -> (('a,'c) 'A))`);

val _ = type_abbrev ("id", Type `: \'A. !'a. ('a ('a 'A))`);

val _ = type_abbrev ("category", Type `: \'A. 'A id # 'A o_arrow`) ;

val category_def = new_definition("category_def",
  ``category = \:'A. \ (id: 'A id, comp: 'A o_arrow).
    (* Left Identity *)
    (!:'a 'b. !(f:'b ('a 'A)). comp id f = f) /\
    (* Right Identity *)
    (!:'a 'b. !(f:'b ('a 'A)). comp f id = f) /\
    (* Composition *)
    (!:'a 'b 'c 'd. !(f:'b ('a 'A)) (g:'c ('b 'A)) (h:'d ('c 'A)).
      comp h (comp g f) = comp (comp h g) f)``) ;

val dual_comp_def = Define
  `dual_comp (comp : 'A o_arrow) : ('A C) o_arrow =
    (\:'c 'b 'a. \f g. comp [:'a,'b,'c:] g f)` ;

category_dual : |- category [:'A:] (id, comp) <=>
  category [:'A C:] (id, dual_comp comp)
```

Note that functors F and G (that is, their action on arrows) are given, in HOL$_\omega$, as `F'` and `G`. We use a prime in `F'` is because `F` is the defined constant "false".

```
val _ = type_abbrev ("g_functor",
  Type `: \'C 'D 'F. !'a 'b. ('a, 'b) 'C -> ('a 'F, 'b 'F) 'D`);

val g_functor_def = Define
  `g_functor = \:'C 'D 'F. \ ((idC, compC) : 'C category)
     ((idD, compD) : 'D category) (F': 'F (('C, 'D) g_functor)).
  (* Identity *) (!:'a. F' [:'a, 'a:] idC = idD) /\
  (* Composition *) (!:'a 'b 'c. !f g.
    F' (compC [:'a,'b,'c:] g f) = compD (F' g) (F' f)) ` ;

val g_dual_functor_def = Define
  `g_dual_functor (F': 'F (('C, 'D) g_functor)) =
```

```
    (\:'a 'b. F' [:'b, 'a:]) : 'F (('C C, 'D C) g_functor)` ;

g_functor_dual :
  |- g_functor [:'C, 'D, 'F:] (idC,compC) (idD,compD) F' <=>
    g_functor [:'C C, 'D C, 'F:] (idC,dual_comp compC)
      (idD,dual_comp compD) (g_dual_functor F')
```

A.2 Definitions and Theorems characterising Adjoint Pairs

We first show the definition of the predicates `g_adjf`n, which are used later. These predicates do not represent the entire conditions for adjoint pairs — they do not include that F, G are functors, that η, ϵ are natural transformations, etc.

```
  |- g_adjf1 (idC,compC) G eta sharp <=>
     !:'a 'b. !f g. (compC (G g) eta = f) <=> (sharp f = g)

  |- g_adjf2 (idD,compD) F' eps flatt <=>
     !:'b 'a. !g f. (compD eps (F' f) = g) <=> (flatt g = f)

  |- g_adjf3 (idC,compC) (idD,compD) F' G eta eps <=>
     !:'a 'b. !f g. (compC (G g) eta = f) <=> (compD eps (F' f) = g)

  |- g_adjf4 (idC,compC) (idD,compD) sharp flatt <=>
     (!:'a 'b. !f g. (flatt g = f) <=> (sharp f = g)) /\
     (!:'a 'c 'b. !h f. sharp (compC f h) =
  compD (sharp f) (sharp (compC (flatt idD) h))) /\
     !:'a 'c 'b. !h g. flatt (compD h g) =
 compC (flatt (compD h (sharp idC))) (flatt g)))

  |- g_adjf5 (idC,compC) (idD,compD) F' G eta eps <=>
     (!:'b. compC (G eps) eta = idC) /\ !:'a. compD eps (F' eta) = idD
```

Here are some examples of some duality theorems, required to use the duality of Theorem 2(i) and (ii), whereas Theorem 2(iii) is self-dual.

```
g_adjf3_dual : |- g_adjf3 [:'C, 'D:] (idC,compC) (idD,compD) F' G eta eps =
    g_adjf3 [:'D C, 'C C:] (idD, dual_comp compD) (idC, dual_comp compC)
  (g_dual_functor G) (g_dual_functor F') eps eta

g_adjf12_dual : |- g_adjf2 [:'D,'C:] (idD,compD) F' eps flatt =
    g_adjf1 [:'D C,'C C:] (idD, dual_comp compD) (g_dual_functor F') eps flatt

SHARP_FLATT_dual : |- SHARP (idD,compD) F' eps =
     FLATT (idD,dual_comp compD) (g_dual_functor F') eps

ETA_EPS_dual : |- ETA (idD,compD) flatt = EPS (idD,dual_comp compD) flatt
```

Here are the theorems justifying Theorem 2. The actual text of all the theorems is rather long, and is omitted, but it is clear from the way we construct the terms to be proved. Note how we use the alternative versions, `cats'` and `taf3'`, to facilitate proving `adjf_thm_2_eq_3` by duality from `adjf_thm_1_eq_3`. Note also how we express the common definition in category theory texts, "…there exists a unique arrow g such that $Gg \circ \eta = f$ …" as "there exists a function $\cdot^\sharp$ such that $Gg \circ \eta = f$ iff $g = f^\sharp$". The final theorem, `g_adjf4_eqv`, expresses that the second and third parts of Theorem 2(iv) are equivalent.

```
val gatm = ``category [:'C:] (idC, compC) ==> category [:'D:] (idD, compD) ==>
  g_adjf1 [:'C, 'D, 'F, 'G:] (idC,compC) G eta sharp ==>
  g_adjf2 [:'D, 'C, 'G, 'F:] (idD,compD) F' eps flatt ==>
  g_adjf3 [:'C, 'D, 'F, 'G:] (idC,compC) (idD,compD) F' G eta eps ==>
  g_adjf4 [:'C, 'D, 'F, 'G:] (idC,compC) (idD,compD) sharp flatt ==>
  (!:'a 'b. !f g. (flatt [:'b,'a:] g = f) <=> (sharp [:'a,'b:] f = g)) ==>
  (!:'a 'c 'b. !h f. sharp [:'a,'b:] (compC f h) =
    compD (sharp [:'c,'b:] f) (sharp (compC (flatt idD) h))) ==>
  (!:'a 'c 'b. !h g. flatt [:'c,'a:] (compD h g) =
    compC (flatt (compD h (sharp idC))) (flatt [:'b,'a:] g)) ==>
  g_adjf5 [:'C, 'D, 'F, 'G:] (idC,compC) (idD,compD) F' G eta eps ==>
  g_functor [:'D,'C,'G:] (idD,compD) (idC,compC) G ==>
  g_functor [:'C,'D,'F:] (idC,compC) (idD,compD) F' ==>
  g_nattransf [:'C:] (idC, compC) eta (g_I [:'C:]) (G g_oo F') ==>
  g_nattransf (idD,compD) eps (F' g_oo G) (g_I [:'D:]) ==>
  (eps = EPS (idC,compC) sharp) ==> (flatt = FLATT (idC,compC) G eta) ==>
  (eta = ETA (idD,compD) flatt) ==> (sharp = SHARP (idD,compD) F' eps) ==>
  (G = \:'a 'b. \g. flatt (compD g eps)) ==>
  (F' = \:'a 'b. \f. sharp (compC eta f)) ==> T `` ;

val ([catC, catD, adjf1, adjf2, adjf3, adjf4,
  adjf4a, adjf4b, adjf4c, adjf5, funG, funF,
  nteta, nteps, eps, flatt, eta, sharp, Geq, Feq], _) = strip_imp gatm ;

(* now to set out our adjoint functor results in the form in the paper *)
val cats = mk_conj (catC, catD) ;
val cats' = mk_conj (catD, catC) ;
val taf1 = list_mk_conj [funG, nteta, adjf1, eps, flatt, Feq] ;
val taf2 = list_mk_conj [funF, nteps, adjf2, eta, sharp, Geq] ;
val taf3 = list_mk_conj [funF, funG, adjf3, flatt, sharp] ;
val taf3' = list_mk_conj [funG, funF, adjf3, sharp, flatt] ;
val taf4 = list_mk_conj [adjf4, eta, eps, Feq, Geq] ;
val taf5 = list_mk_conj [funF, funG, nteta, nteps, adjf5, flatt, sharp] ;

val adjf_thm_1_eq_3 = store_thm ("adjf_thm_1_eq_3",
  mk_imp (cats, mk_eq (taf1, taf3)), ...) ;
```

```
val adjf_thm_2_eq_3 = store_thm ("adjf_thm_2_eq_3",
  mk_imp (cats', mk_eq (taf2, taf3')), ...) ;

val adjf_thm_3_eq_4 = store_thm ("adjf_thm_3_eq_4",
  mk_imp (cats, mk_eq (taf3, taf4)), ...) ;

val adjf_thm_3_eq_5 = store_thm ("adjf_thm_3_eq_5",
  mk_imp (cats, mk_eq (taf3, taf5)), ...) ;

(* to show the second and third parts of (iv) are equivalent *)
val g_adjf4_eqv = store_thm ("g_adjf4_eqv",
  list_mk_imp ([cats, adjf4a], mk_eq (adjf4b, adjf4c)), ...) ;
```

A.3 Definitions and Theorems about Monads

Here is the definition of the Kleisli arrow type abbreviation.

```
val _ = type_abbrev ("Kleisli", Type `: \'A 'M 'a 'b. ('a, 'b 'M) 'A`) ;
```

The combination of postfix type constructors, and the fact that the conventional notation $(\alpha, \beta)C$ is not syntactic sugar for a curried application of a higer order type constructor C has led to the situation where the following are all equivalent:

```
 ('A, 'M, 'a, 'b) Kleisli
 ('a, 'b) (('A, 'M) Kleisli)
 'b ('a ('M ('A Kleisli)))
```

We have several predicates defining, in various ways, a monad.

`Komonad`**:** rules (E1) to (E3)
`Kmonad`**:** rules (E1), (E2) and a variant of (E3) obtained by substituting (E6)
`Kdomonad`**:** rules (E1) to (E5)
`Kdmonad`**:** as for `Kmonad`, plus rules (E4) and (E5)
`g_umj_monad`**:** the *definition* of this is explained in Appendix A.11, but it is *proved* equivalent to rules (1) to (7) (in `g_umj_monad_exp`)

As noted in the main text, rules (E1) to (E3) imply rule (E6), and so we have the theorem `Komonad_iff`.

```
Kcomp_def : |- !id comp ext.
  Kcomp (id,comp) ext = (\:'a 'b 'c. (\h k. comp (ext h) k))

Komonad_iff : |- category (id,comp) ==>
    (Komonad (id,comp) (unit,ext,kcomp) <=>
    Kmonad (id,comp) (unit,ext) /\ (kcomp = Kcomp (id,comp) ext))
```

The pair of adjoint functors to and from the Kleisli category of a monad: the adjoint functors are *ext* and the functor which is the identity on objects, and takes arrow f to $unit \circ f$.

```
Komonad_IMP_uof :
    |- category (id,comp) /\ Komonad (id,comp) (unit,ext,kcomp) ==>
          g_functor (id,comp) (unit,kcomp) (\:'a 'b. comp unit)

Komonad_IMP_ext_f :
    |- Komonad (id,comp) (unit,ext,kcomp) ==>
          g_functor (unit,kcomp) (id,comp) ext
```

The following theorems show that these functors form an adjoint pair: they show respectively that *ext* is a right adjoint, and that *unit* is a natural transformation.

```
Kmonad_IMP_adjf;
    |- Kmonad (id,comp) (unit,ext) ==> g_adjf1 (id,comp) ext unit (\:'a 'b. I)

Kmonad_IMP_unit_nt;
    |- Kmonad (id,comp) (unit,ext) ==>
       g_nattransf (id,comp) unit g_I (\:'a 'b. (\f. ext (comp unit f)))
```

Finally, every pair of adjoint functors F and G gives a monad, whose action on objects is $G \circ F$, whose *unit* (η) is just η (!), and where *ext* f is $G(f^\sharp)$

```
g_adjf1_IMP_Kmonad :
  |- category (idC,compC) /\ g_functor (idD,compD) (idC,compC) G /\
          g_adjf1 (idC,compC) G eta sharp ==>
Kmonad (idC,compC) (eta,(\:'a 'b. G o sharp))
```

A.4 Definitions and Theorems about Compound Monads

Here are the rules (1K) to (8K), which are the counterparts, for the monad N_M in $\mathcal{C}_M$, the Kleisli category for M, of rules (1) to (8).

$$
\begin{aligned}
\mathit{kmap}\ \mathit{unit}_M &= \mathit{unit}_M && \text{(1K)}\\
\mathit{kmap}\ f \odot_M \mathit{kmap}\ g &= \mathit{kmap}\ (f \odot_M g) && \text{(2K)}\\
\mathit{unit}_{MN} \odot_M f &= \mathit{kmap}\ f \odot_M \mathit{unit}_{MN} && \text{(3K)}\\
\mathit{kjoin} \odot_M \mathit{kmap}\ (\mathit{kmap}\ f) &= \mathit{kmap}\ f \odot_M \mathit{kjoin} && \text{(4K)}\\
\mathit{kjoin} \odot_M \mathit{unit}_{MN} &= \mathit{unit}_M && \text{(5K)}\\
\mathit{kjoin} \odot_M \mathit{kmap}\ \mathit{unit}_{MN} &= \mathit{unit}_M && \text{(6K)}\\
\mathit{kjoin} \odot_M \mathit{kmap}\ \mathit{kjoin} &= \mathit{kjoin} \odot_M \mathit{kjoin} && \text{(7K)}
\end{aligned}
$$

$$
\mathit{pext}\ f = kjoin \odot_M \mathit{kmap}\ f \qquad \text{(8K)}
$$

Theorems 5 and 6 are combined in the single HOL theorem `cm_Ko_J1S`.

```
J1S_def : |- !id comp extM extNM.
   J1S (id,comp) extM extNM <=>
```

```
  !:'a 'b. !f. extM (extNM f) = comp (extNM f) (extM id)

cm_Ko_J1S :
  |- category (id,comp) /\ Komonad (id,comp) (unitM,extM,kcomp) ==>
     (Komonad (id,comp) (unitNM,extNM,oNM) /\ J1S (id,comp) extM extNM /\
       (pext = (\:'a 'b. (\f. comp (extNM f) unitM))) <=>
     Komonad (unitM,kcomp) (unitNM,pext,oNM) /\
       (extNM = (\:'a 'b. (\f. extM (pext f)))))
```

A.5 Theorems about the Compatibility Conditions

(C2) $\Rightarrow$ (C5): holds even if $map_M join_N$ is replaced by an arbitrary term `djoin`

```
BW_C2_C5: |- category (id,comp) /\ Kmonad (id,comp) (unitNM,extNM) ==>
       (extNM unitM = djoin) ==> (comp djoin (extNM id) = extNM djoin)
```

(C5) $\Rightarrow$ (C2):

```
BW_C5_C2 : |- category (id,comp) ==> Kmonad (id,comp) (unitNM,extNM) ==>
  Kdmonad (id,comp) (unitN,extN,mapN,joinN) ==>
  Kdmonad (id,comp) (unitM,extM,mapM,joinM) ==>
  (unitNM = (\:'b. comp unitM unitN)) ==>
  (comp (mapM joinN) (extNM id) = extNM (mapM joinN)) ==>
  (extNM unitM = mapM joinN)
```

(C3) $\Leftrightarrow$ (C4): For these, we first showed that each of (C3) and (C4) are equivalent to their more general versions (recall that (C4) is just J(1)).

```
C3_IFF_C3S:
  |- category (id,comp) /\ Kdmonad (id,comp) (unitM,extM,mapM,joinM) /\
       Kdmonad (id,comp) (unitN,extN,mapN,joinN) /\
       Kdmonad (id,comp) (unitNM,extNM,mapNM,joinNM) /\
       (mapNM = (\:'a 'b. (\f. mapM (mapN f)))) ==>
     ((!:'a. comp joinNM (mapM unitN) = joinM) <=>
     !:'a 'b. !f. comp (extNM f) (mapM unitN) = extM f)

C4_IFF_J1S:
  |- category (id,comp) /\ Kdmonad (id,comp) (unitM,extM,mapM,joinM) /\
       Kdmonad (id,comp) (unitNM,extNM,mapNM,joinNM) /\
       (!:'a 'b. !f. extM (mapNM f) = comp (mapNM f) joinM) ==>
     ((!:'a. extM joinNM = comp joinNM joinM) <=>
     J1S (id,comp) extM extNM)
```

Then we show that the more general versions are equivalent.

```
J1S_IFF_C3S :
  |- category (id,comp) /\ Kdmonad (id,comp) (unitM,extM,mapM,joinM) /\
```

```
  Kmonad (id,comp) (unitNM,extNM) /\
  (unitNM = (\:'a. comp unitM unitN)) ==>
(J1S (id,comp) extM extNM <=>
!:'a 'b. !f. comp (extNM f) (mapM unitN) = extM f)
```

A.6 Duality of proofs: (C2) ⇔ (C5) and (C3) ⇔ (C4)

We indicate the correspondence between the category theory notation and the notation of this paper.

$$
\begin{array}{llr}
\mu & join_{MN} : \alpha NMNM \to \alpha NM & \\
\mu F & join_{MN} : \alpha FNMNM \to \alpha FNM & \\
\mu, \mu_N, \mu_M & join_{MN}, join_N, join_M & \\
\eta, \eta_N, \eta_M & unit_{MN}, unit_N, unit_M & \\
T, N, M & map_{MN}, map_N, map_M & \\
T = MN & map_{MN} = map_M \circ map_N & \text{(MC)} \\
\mu \circ \eta T = id & join \circ unit = id & (5) \\
\mu \circ T\eta = id & join \circ map\ unit = id & (6) \\
\mu \circ T\mu = \mu \circ \mu T & join \circ map\ join = join \circ join & (7)
\end{array}
$$

We first list the conditions (C1) to (C5).

$$
\begin{array}{lr}
\eta = M\eta_N \circ \eta_M = \eta_M N \circ \eta_N & \text{(C1)} \\
\mu \circ MN\eta_M N = M\mu_N & \text{(C2)} \\
\mu \circ M\eta_N MN = \mu_M N & \text{(C3)} \\
\mu_M N \circ M\mu = \mu \circ \mu_M NMN & \text{(C4)} \\
M\mu_N \circ \mu N = \mu \circ MNM\mu_N & \text{(C5)}
\end{array}
$$

Finally we show the proofs of equivalence. Note the duality between the notations of the proofs. Steps with no reasons given rely on either M or N being functors ($T(\phi \circ \psi) = T\phi \circ T\psi$) or on the definition of composition of natural transformations ($(\phi \circ \psi)T = \phi T \circ \psi T$). Observe how the duality between the proofs also uses the duality between rules (5) and (6), and the self-duality of rule (7) (see the table above). But also note the duality between the specific uses made of the naturality of η_N and of μ.

(C4) ⇒ (C3):

$$
\begin{array}{rlr}
\mu \circ M\eta_N MN & = \mu \circ (\mu_M \circ M\eta_M)\, NMN \circ M\eta_N MN & (6 \text{ for } M) \\
& = \mu \circ \mu_M NMN \circ M\eta_M NMN \circ M\eta_N MN & \\
& = \mu \circ \mu_M NMN \circ M\,(\eta_M N \circ \eta_N)\, MN & \\
& = \mu \circ \mu_M NMN \circ M\eta MN & \text{(C1)} \\
& = \mu_M N \circ M\mu \circ M\eta MN & \text{(C4)} \\
& = \mu_M N \circ M\,(\mu \circ \eta MN) = \mu_M N & (5 \text{ for } MN)
\end{array}
$$

(C3) $\Rightarrow$ (C4):

$$\begin{aligned}
\mu_M N \circ M\mu &= \mu \circ M\eta_N MN \circ M\mu && \text{(C3)}\\
&= \mu \circ M\,(\eta_N MN \circ \mu)\\
&= \mu \circ M\,(N\mu \circ \eta_N MNMN) && (\eta_N \text{ natural})\\
&= \mu \circ MN\mu \circ M\eta_N MNMN\\
&= \mu \circ \mu MN \circ M\eta_N MNMN && (7 \text{ for } MN)\\
&= \mu \circ \mu_M NMN && \text{(C3)}
\end{aligned}$$

(C5) $\Rightarrow$ (C2):

$$\begin{aligned}
\mu \circ MN\eta_M N &= \mu \circ MNM\,(\mu_N \circ \eta_N N)\ \circ MN\eta_M N && (5 \text{ for } N)\\
&= \mu \circ MNM\mu_N \circ MNM\eta_N N \circ MN\eta_M N\\
&= \mu \circ MNM\mu_N \circ MN\,(M\eta_N \circ \eta_M)\,N\\
&= \mu \circ MNM\mu_N \circ MN\eta N && \text{(C1)}\\
&= M\mu_N \circ \mu N \circ MN\eta N && \text{(C5)}\\
&= M\mu_N \circ\ (\mu \circ MN\eta)\,N = M\mu_N && (6 \text{ for } MN)
\end{aligned}$$

(C2) $\Rightarrow$ (C5):

$$\begin{aligned}
M\mu_N \circ \mu N &= \mu \circ MN\eta_M N \circ \mu N && \text{(C2)}\\
&= \mu \circ\ (MN\eta_M \circ \mu)\,N\\
&= \mu \circ\ (\mu M \circ MNMN\eta_M)\,N && (\mu \text{ natural})\\
&= \mu \circ \mu MN \circ MNMN\eta_M N\\
&= \mu \circ MN\mu \circ MNMN\eta_M N && (7 \text{ for } MN)\\
&= \mu \circ MNM\mu_N && \text{(C2)}
\end{aligned}$$

A.7 Barr & Wells (D3) and (D4) versus Jones & Duponcheel S(4)

Recall that

- Barr & Wells defined a distributive law to be a natural transformation λ saisfying their (D1) to (D4)
- Barr & Wells defined a compound monad satisfying (MC) to satisfy the compatibility condition if their (C1) to (C5) are satisfied
- Barr & Wells showed that compatibility and the existence of a distributive law to be equivalent
- Jones & Duponcheel assumed (MC) and (UC) throughout, and showed that the existence of a function *swap* (ie, λ) satisfying their conditions S(1) to S(4) is equivalent to a compound monad satisfying J(1) and J(2)
- S(1) is that *swap* is natural, (C1) is (UC), (C4) and (C5) are J(1) and J(2), and (D1) and (D2) are S(3) and S(2)
- We showed that (C3) and (C2) are equivalent to (C4) and (C5) respectively, and so to J(1) and J(2)

Therefore we should be able to prove (D3) and (D4) from S(4), and vice versa.

$$\begin{aligned} \mathit{swap} \circ \mathit{map}_N\ \mathit{unit}_M &= \mathit{unit}_M \\ \lambda \circ N\eta_M &= \eta_M N \end{aligned} \tag{D1}$$

$$\begin{aligned} \mathit{swap} \circ \mathit{unit}_N &= \mathit{map}_M\ \mathit{unit}_N \\ \lambda \circ \eta_N M &= M\eta_N \end{aligned} \tag{D2}$$

$$\begin{aligned} \mathit{swap} \circ \mathit{map}_N\ \mathit{join}_M &= \mathit{ext}_M\ \mathit{swap} \circ \mathit{swap} \\ \lambda \circ N\mu_M &= \mu_M N \circ M\lambda \circ \lambda M \end{aligned} \tag{D3}$$

$$\begin{aligned} \mathit{swap} \circ \mathit{join}_N &= \mathit{map}_M\ \mathit{join}_N \circ \mathit{swap} \circ \mathit{map}_N\ \mathit{swap} \\ \lambda \circ \mu_N M &= M\mu_N \circ \lambda N \circ N\lambda \end{aligned} \tag{D4}$$

$$\mathit{prod} = \mathit{map}_M\ \mathit{join}_N \circ \mathit{swap} \qquad \mathit{dorp} = \mathit{ext}_M\ \mathit{swap}$$

$$\mathit{prod} \circ \mathit{map}_N\ \mathit{dorp} = \mathit{dorp} \circ \mathit{prod} \tag{S(4)}$$

To prove (D4) from S(4), do $\ldots \circ \mathit{map}_N\ \mathit{unit}_M$ to both sides. Then lhs gives $\mathit{prod} \circ \mathit{map}_N\ (\mathit{dorp} \circ \mathit{unit}_M) = \mathit{prod} \circ \mathit{map}_N\ \mathit{swap}$, ie, the rhs of (D4). Using (D1), $\mathit{prod} \circ \mathit{map}_N\ \mathit{unit}_M = \mathit{map}_M\ \mathit{join}_N \circ \mathit{unit}_M = \mathit{unit}_M \circ \mathit{join}_N$, so $\mathit{dorp} \circ \mathit{prod} \circ \mathit{map}_N\ \mathit{unit}_M = \mathit{ext}_M\ \mathit{swap} \circ \mathit{unit}_M \circ \mathit{join}_N = \mathit{swap} \circ \mathit{join}_N$, is, the lhs of (D4).

An argument which is actually dual in the sense described earlier gives (D3): it involves doing $\ldots \circ \mathit{map}_N\ (\mathit{map}_M\ \mathit{unit}_N)$. (It uses (D2)). From rhs of S(4), we get the rhs of (D3).

$$\begin{aligned} &\mathit{dorp} \circ \mathit{prod} \circ \mathit{map}_N\ (\mathit{map}_M\ \mathit{unit}_N) \\ &\quad = \mathit{dorp} \circ \mathit{map}_M\ \mathit{join}_N \circ \mathit{swap} \circ \mathit{map}_N\ (\mathit{map}_M\ \mathit{unit}_N) \\ &\quad = \mathit{dorp} \circ \mathit{map}_M\ \mathit{join}_N \circ \mathit{map}_M\ (\mathit{map}_N\ \mathit{unit}_N) \circ \mathit{swap} \\ &\quad = \mathit{dorp} \circ \mathit{map}_M\ (\mathit{join}_N \circ \mathit{map}_N\ \mathit{unit}_N) \circ \mathit{swap} \\ &\quad = \mathit{ext}_M\ \mathit{swap} \circ \mathit{swap} \end{aligned}$$

From lhs of S(4), we get the lhs of (D3).

$$\begin{aligned} &\mathit{prod} \circ \mathit{map}_N\ \mathit{dorp} \circ \mathit{map}_N\ (\mathit{map}_M\ \mathit{unit}_N) \\ &\quad = \mathit{prod} \circ \mathit{map}_N\ (\mathit{ext}_M\ \mathit{swap} \circ \mathit{map}_M\ \mathit{unit}_N) \\ &\quad = \mathit{prod} \circ \mathit{map}_N\ (\mathit{ext}_M\ (\mathit{swap} \circ \mathit{unit}_N)) \\ &\quad = \mathit{prod} \circ \mathit{map}_N\ (\mathit{ext}_M\ (\mathit{map}_M\ \mathit{unit}_N)) \\ &\quad = \mathit{prod} \circ \mathit{map}_N\ (\mathit{map}_M\ \mathit{unit}_N \circ \mathit{join}_M) \\ &\quad = \mathit{prod} \circ \mathit{map}_N\ (\mathit{map}_M\ \mathit{unit}_N) \circ \mathit{map}_N\ \mathit{join}_M \end{aligned}$$

which is the lhs of (D3) because

$$\begin{aligned} \mathit{prod} \circ \mathit{map}_N\ (\mathit{map}_M\ \mathit{unit}_N) &= \mathit{map}_M\ \mathit{join}_N \circ \mathit{swap} \circ \mathit{map}_N\ (\mathit{map}_M\ \mathit{unit}_N) \\ &= \mathit{map}_M\ \mathit{join}_N \circ \mathit{map}_M\ (\mathit{map}_N\ \mathit{unit}_N) \circ \mathit{swap} \\ &= \mathit{map}_M\ (\mathit{join}_N \circ \mathit{map}_N\ \mathit{unit}_N) \circ \mathit{swap} = \mathit{swap} \end{aligned}$$

For the converse, using (D3) and (D4) to prove S(4), we have

$$
\begin{aligned}
& prod \circ map_N \ dorp \\
& = map_M \ join_N \circ swap \circ map_N \ (join_M \circ map_M \ swap) \\
& = map_M \ join_N \circ swap \circ map_N \ join_M \circ map_N \ (map_M \ swap) \\
& = map_M \ join_N \circ join_M \circ map_M \ swap \circ swap \circ map_N \ (map_M \ swap) \\
& = map_M \ join_N \circ join_M \circ map_M \ swap \circ map_M \ (map_N \ swap) \circ swap \\
& = join_M \circ map_M \ (map_M \ join_N) \circ map_M \ swap \circ map_M \ (map_N \ swap) \circ swap \\
& = join_M \circ map_M \ (map_M \ join_N \circ swap \circ map_N \ swap) \circ swap \\
& = join_M \circ map_M \ (swap \circ join_N) \circ swap \\
& = ext_M \ swap \circ prod = dorp \circ prod
\end{aligned}
$$

The proofs above were done in HOL_ω. The predicate `dist_law` says that *swap* is a natural transformation and that (D1) to (D4) hold.

```
S_IFF_DL : |- category (id,comp) ==>
   Kdmonad (id,comp) (unitM,extM,mapM,joinM) ==>
   g_umj_monad (id,comp) (unitN,mapN,joinN) ==>
   (S1to4 (id,comp) (unitM,extM,mapM,joinM) (unitN,mapN,joinN) swap <=>
   dist_law (id,comp) (unitM,extM,mapM,joinM) (unitN,mapN,joinN) swap)
```

A.8 Proof that a $\mathcal{C}_M$ gives a natural transformation *swap*

Note that it follows easily from (E6) that $h \odot (g \circ f) = (h \odot g) \circ f$, so we omit the parentheses in such an expression.

$$
\begin{aligned}
& swap \circ map_N \ (map_M \ f) \\
& = kmap \ (ext_M \ (unit_M \circ f)) \\
& = kmap \ ((unit_M \circ f) \odot_M \ id) \\
& = kmap \ (unit_M \circ f) \odot_M \ kmap \ id \\
& = pext \ (unit_{MN} \odot_M \ unit_M \circ f) \odot_M \ swap \\
& = pext \ (unit_{MN} \circ f) \odot_M \ swap \\
& = ext_M (pext \ (unit_{MN} \circ f)) \circ swap \\
& = ext_{MN} (unit_{MN} \circ f) \circ swap = map_{MN} \ f \circ swap
\end{aligned}
$$

A.9 Compatibility conditions imply a distributive law: further details

Assuming the compatibility condition (C4), ie (J1), we have the monad N_M in $\mathcal{C}_M$. Next, if we assume also the compatibility condition (C2), ie (J2), then, using (PE), (C2) and (3M), we can show $kjoin = unit_M \circ join_N$.

$$kjoin = pext \ unit_M = ext_{MN} \ unit_M \circ unit_M = map_M \ join_N \circ unit_M = unit_M \circ join_N$$

We use the definition of the distributive law λ [1, §9.2], ie, *swap* [6, §4.3], namely $swap = pext\ (map_M\ unit_N)$. We show this is equal to $kmap\ id$.

$$\begin{aligned} swap = pext\ (map_M\ unit_N) &= pext\ (ext_M(unit_M \circ unit_N)) \\ &= pext\ (unit_{MN} \odot_M\ id) = kmap\ id \end{aligned}$$

A.10 Proof that a distributive law gives a monad in $\mathcal{C}_M$

Here we are given *swap* satisfying (D1) to (D4), and we *define*

$$\begin{aligned} kjoin &= unit_M \circ join_N \\ kmap\ f &= swap \circ map_N\ f \\ pext\ f &= kjoin \odot_M\ kmap\ f \\ unit_{MN} &= unit_M \circ unit_N \\ map_{MN}\ f &= map_M\ (map_N\ f) \end{aligned}$$

We show (1K) to (7K), and also J(2). Note that $kjoin \odot_M f = map_M\ join_N \circ f$, and $kjoin \odot_M\ unit_M \circ g = unit_M \circ join_N \circ g$.

(1K) is just (D1), and $kmap\ (unit_M \circ f) = unit_M \circ map_N\ f$. Thus $kmap\ unit_{MN} = unit_M \circ map_N\ unit_N$ and $kmap\ kjoin = unit_M \circ map_N\ join_N$.

Now (5K) to (7K) use (5N) to (7N) respectively.

So for (5K), $kjoin \odot_M\ unit_{MN} == unit_M \circ join_N \circ unit_N = unit_M$

For (6K), $kjoin \odot_M kmap\ unit_{MN} == unit_M \circ join_N \circ map_N\ unit_N = unit_M$

For (7K), $kjoin \odot_M kjoin = unit_M \circ join_N \circ join_N$ and $kjoin \odot_M kmap\ kjoin = kjoin \odot_M\ (unit_M \circ map_N\ join_N) = unit_M \circ join_N \circ map_N\ join_N$.

For (3K), $kmap\ f \odot_M unit_{MN} = kmap\ f \circ unit_N = swap \circ map_N\ f \circ unit_N = swap \circ unit_N \circ f = map_M\ unit_N \circ f = unit_{MN} \odot_M\ f$

For (4K), using (D4) and (4N),

$$\begin{aligned} kjoin \odot_M\ kmap\ (kmap\ f) &= map_N\ join_N \circ swap \circ map_N\ (swap \circ map_N\ f) \\ &= map_N\ join_N \circ swap \circ map_N\ (swap \circ map_N\ f) \\ &= swap \circ join_N \circ map_N\ (map_N\ f) \\ &= swap \circ map_N\ f \circ join_N \\ &= kmap\ f \odot_M\ unit_M \circ join_N = kmap\ f \odot_M\ kjoin \end{aligned}$$

For (2K), using (D3) and naturality of *swap*:

$$\begin{aligned} kmap\ g \odot_M\ kmap\ f &= ext_M\ (swap \circ map_N\ g) \circ swap \circ map_N\ f \\ &= ext_M\ swap \circ map_M(map_N\ g) \circ swap \circ map_N\ f \\ &= ext_M\ swap \circ swap \circ map_N(map_M\ g) \circ map_N\ f \\ &= swap \circ map_N\ join_M \circ map_N(map_M\ g) \circ map_N\ f \\ &= swap \circ map_N\ (ext_M\ g \circ f) = kmap\ (g \odot_M\ f) \end{aligned}$$

We have proved this result HOL_ω. Recall that the predicate `dist_law` says that *swap* is a natural transformation and that (D1) to (D4) hold.

```
BWD_cm : |- category (id,comp) ==>
   Kdomonad (id,comp) (unitM,extM,kcomp,mapM,joinM) ==>
   g_umj_monad (id,comp) (unitN,mapN,joinN) ==>
   dist_law (id,comp) (unitM,extM,mapM,joinM) (unitN,mapN,joinN) swap ==>
   (kjoin = (\:'a. comp unitM joinN)) ==>
   (kmap = (\:'a 'b. (\f. comp swap (mapN f)))) ==>
   (unitNM = (\:'a. comp unitM unitN)) ==>
   g_umj_monad (unitM,kcomp) (unitNM,kmap,kjoin)
```

We now have proved (1K) to (7K), and our definition of *pext* is (8K) so we can now use (E4K), ie $pext\ unit_M = kjoin$. Then we can prove (C2), ie (J2), assuming that ext_{MN} is defined by (PE):

$$ext_{MN}\ unit_M = ext_M\ (pext\ unit_M) = ext_M\ (unit_M \circ join_N) = map_M\ join_N$$

Now we also want to show that when we define the monad MN by ext_{MN} and $\odot_{MN}$, as in Theorem 6, then the definition of map_{MN} that results is equivalent to (MC). To show this, note that $kmap\ (g \circ f) = kmap\ g \circ map_N\ f$ by definition, and that $pext\ (g \circ f) = pext\ g \circ map_N\ f$ can be proved from it. Then

$$\begin{aligned} ext_M\ (pext\ (unit_{MN} \circ f)) &= ext_M\ (pext\ unit_{MN} \circ map_N\ f) \\ &= ext_M\ (unit_M \circ map_N\ f) = map_M\ (map_N\ f) \end{aligned}$$

So, from the premises of `BWD_cm` we also get, in theorems `BWD_cmK` and `BWD_cmKJM`: (premises as in `BWD_cm` shown as `...`)

```
BWD_cmK : |- ... ==>
   Kdmonad (unitM,kcomp) (unitNM,pext,kmap,kjoin)

BWD_cmKJM : |- ... ==>
  Kdmonad (unitM,kcomp) (unitNM,pext,kmap,kjoin) ==>
  (extM (pext unitM) = mapM joinN) /\
  (extM (pext (comp unitNM f)) = mapM (mapN f))
```

A.11 Generalising the 7-rule axiom system to Compound Monads

We now present a generalisation of the system of axioms (1) to (8), and their equivalence to (E1) to (E5). This was motivated by the construction by Jones & Duponcheel [6, §3.3] using their function *dorp*.

We found that the rules (G1) to (G8), which are analogous to rules (1) to (8), are sufficient to establish that NM is a monad, without assuming that either N or M is even a premonad. In these rules, we make no assumptions about the functions we call ext_{MN}, $join_{MN}$, map_{MN}, $unit_{MN}$ or $unit_M$.

These rules also use three more functions of the following types:

$$\begin{aligned} dunit &: \alpha M \to \alpha NM \\ dmap &: (\alpha \to \beta M) \to (\alpha NM \to \beta NM) \\ djoin &: \alpha NNM \to \alpha NM \end{aligned}$$

$$dmap\ unit_M = id \tag{G1}$$
$$dmap\ (f \circ h) = dmap\ f \circ map_{MN}\ h \tag{G2}$$
$$dmap\ f \circ unit_{MN} = dunit \circ f \tag{G3}$$
$$djoin \circ dmap\ (dmap\ f) = dmap\ f \circ join_{MN} \tag{G4}$$
$$djoin \circ dunit = id \tag{G5}$$
$$djoin \circ dmap\ unit_{MN} = id \tag{G6}$$
$$djoin \circ dmap\ djoin = djoin \circ join_{MN} \tag{G7}$$

$$ext_{MN}\ f = djoin \circ dmap\ f \tag{G8}$$

Now we can prove analogues of the equivalence of the various monad presentations in §3. In fact, the proofs are very similar to the proofs for a single monad.

Theorem 9. *Assume rules (G1) to (G8). Then ext_{MN}, $join_{MN}$, map_{MN} and $unit_{MN}$ give a monad NM, where also*

$$djoin = ext_{MN}\ unit_M \tag{G9}$$
$$dmap\ f = ext_{MN}\ (dunit \circ f) \tag{G10}$$
$$unit_{MN} = dunit \circ unit_M \tag{G11}$$
$$map_{MN}\ f = dmap\ (unit_M \circ f) \tag{G12}$$

The following converse result tells us when a monad NM can be defined in this way. Note that we are still not assuming that M or N is a monad.

Theorem 10. *Assume that NM is a monad. Also assume that rules (G5) and (G9) to (G11) hold. Then the remaining rules among (G1) to (G8) hold.*

In the HOL$_\omega$ implementation, `Gmonad` refers to rules (G1) to (G7). Thus in HOL$_\omega$ the equivalence `Gmonad_iff_Kmonad` represents Theorems 9 and 10.

```
EXTD_def : |- !id comp dmap djoin.
   EXTD (id,comp) (dmap,djoin) = (\:'a 'b. (\f. comp djoin (dmap f)))

Gmonad_iff_Kmonad : |- category (id,comp) ==>
  (Gmonad (id,comp) (dunit,dmap,djoin) (unitNM,mapNM,joinNM) unitM /\
  (extNM = EXTD (id,comp) (dmap,djoin)) <=>
  Kdmonad (id,comp) (unitNM,extNM,mapNM,joinNM) /\
  (djoin = (\:'a. extNM unitM)) /\
  (dmap = (\:'a 'b. (\f. extNM (comp dunit f)))) /\
  (unitNM = (\:'a. comp dunit unitM)) /\ !:'a. comp djoin dunit = id)
```

Given a monad NM, we consider when it can be constructed using Theorem 9, now assuming that M and N are monads. Now, (G9) and (G10) are just definitions, and we define $dunit = map_M\ unit_N$ to satisfy (G11). So the only condition is that (G5) holds, which is now $ext_{MN}\ unit_M \circ map_M\ unit_N = id$. As we noted earlier, *either* J(1) *or* J(2) implies this identity.

The function *dorp* of Jones & Duponcheel [6] is just $dorp = dmap\ id$.

Now if we set M to be the identity monad, then $unit_M$ becomes the identity, *dunit*, *dmap* and *djoin* become $unit_N$, map_N and $join_N$, and $unit_{MN}$, map_{MN}, $join_{MN}$ and ext_{MN} become $unit_N$, map_N, $join_N$ and ext_N.

In fact in the HOL_ω development, we *defined* `g_umj_monad` in terms of `Gmonad` (`g_umj_monad_def`), and then *proved* that it corresponds to the usual statement of the 7 rules, in `g_umj_monad_exp`.

```
g_umj_monad_exp : |- g_umj_monad (id,comp) (unit,map,join) <=>
   (!:'a. map id = id) /\
   (!:'a 'b 'c. !f g. map (comp g f) = comp (map g) (map f)) /\
   (!:'a 'b. !f. comp (map f) unit = comp unit f) /\
   (!:'a 'b. !f. comp (map f) join = comp join (map (map f))) /\
   (!:'a. comp join unit = id) /\
   (!:'a. comp join (map unit) = id) /\
   !:'a. comp join (map join) = comp join join

g_umj_monad_def :
  |- g_umj_monad = (\:'A 'N. (\(id,comp) (unitN,mapN,joinN).
     Gmonad (id,comp) (unitN,mapN,joinN) (unitN,mapN,joinN) id))
```

Then Theorem 9 gives that rules (1) to (7), with (8) and (E6) as definitions, imply rules (E1) to (E3) and the converse definitions (E4) and (E5).

To use the converse Theorem 10 in proving the converse direction of Theorem 3, we would need to first prove (G5), ie, $join_N \circ unit_N = id$, which is easy — it is just (E2), and also (E6), and then Theorem 10 gives the rest.

```
EXT_def : |- !id comp map join.
   EXT (id,comp) (map,join) = (\:'a 'b. (\f. comp join (map f)))

g_umj_iff_Kdomonad : |- category (id,comp) ==>
  (g_umj_monad (id,comp) (unit,map,join) /\
    (ext = EXT (id,comp) (map,join)) /\
    (kcomp = Kcomp (id,comp) ext) <=>
  Kdomonad (id,comp) (unit,ext,kcomp,map,join))
```