Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Effect polymorphism in higher-order logic (proof pearl)

Andreas Lochbihler

the date of receipt and acceptance should be inserted later

Received: date / Accepted: date

Abstract The notion of a *monad* cannot be expressed within higher-order logic (HOL) due to type system restrictions. I show that if a monad is restricted to values of a fixed type, this notion *can* be formalised in HOL. Based on this idea, I develop a library of effect specifications and implementations of monads and monad transformers. Hence, I can abstract over the concrete monad in HOL definitions and thus use the same definition for different (combinations of) effects. I illustrate the usefulness of effect polymorphism with a monadic interpreter.

Keywords monad · monad transformer · effects · polymorphism · equational reasoning · Isabelle/HOL

1 Introduction

Monads have become a standard way to write effectful programs in pure functional languages [34]. In proof assistants, they provide a popular abstraction for modelling and reasoning about effects [4,5,8,19,22,32,36]. Abstractly, a monad consists of a type constructor τ and two polymorphic operations, $\mathsf{return} :: \alpha \Rightarrow \alpha\ \tau$ for embedding values and $\mathsf{bind} :: \alpha\ \tau \Rightarrow (\alpha \Rightarrow \beta\ \tau) \Rightarrow \beta\ \tau$ for sequencing, with infix notation $\gg\!=$, satisfying three monad laws:

1. $(m \gg\!= f) \gg\!= g = m \gg\!= (\lambda x.\ f\ x \gg\!= g)$
2. $\mathsf{return}\ x \gg\!= f \quad = f\ x$
3. $m \gg\!= \mathsf{return} \qquad = m$

Yet, the notion of a monad cannot be expressed as a formula in higher-order logic (HOL) [12] as there are no type constructor variables like τ in HOL and the

This article extends the conference version presented at *Interactive Theorem Proving* 2017 [23]. Most of this work was done while the author was at the Institute of Information Security at ETH Zurich, Zurich, Switzerland.

A. Lochbihler
Digital Asset (Switzerland) GmbH
Luggwegstrasse 9, CH-8048 Zurich
E-mail: mail@andreas-lochbihler.de

sequencing operation bind occurs with three different type instances in the first law. Thus, only concrete monad instances have been used to model side effects of HOL functions. In fact, monad definitions for different effects abound in HOL, e.g., a state-error monad [4], memoization [36], non-determinism with errors and divergence [19], probabilistic choice [5], probabilistic resumptions with errors [22], and certification monads [32]. Each of these formalisations fixes τ to a particular type (constructor) and develops its own reasoning infrastructure. This approach achieves *value polymorphism*, i.e., one monad can be used with varying types of values, but not *effect polymorphism* where one function can be used with different monads.

In this article, I give up value polymorphism in favour of effect polymorphism. The idea is to fix the type of values to some type α_0. Then, the monad type constructor τ is applied only to α_0. The resulting combination $\alpha_0\ \tau$ is represented by an ordinary HOL type variable μ. So, the monad operations have the HOL types $\mathsf{return} :: \alpha_0 \Rightarrow \mu$ and $\mathsf{bind} :: \mu \Rightarrow (\alpha_0 \Rightarrow \mu) \Rightarrow \mu$. This notion of a monad can be formalised within HOL. In detail, I present an Isabelle/HOL library[1] for different monadic effects and their algebraic specification. All effects are also implemented as value-monomorphic monads and monad transformers. Using Isabelle's module system [2], function definitions can be made abstractly and later specialised to several concrete monads. As a running example, I formalise and reason about a monadic interpreter for a small arithmetic language. The library has been used in a larger project to define and reason about parsers and serialisers for security protocols (§6).

Contributions. I show the advantages of trading in value polymorphism for effect polymorphism. First, HOL functions with effects can be defined in an abstract monadic setting (§2) and reasoned about in the style of Gibbons and Hinze [7]. This preserves the level of abstraction that the monad notion provides. As the definitions need not commit to a concrete monad, they can be used in richer effect contexts too—simply by combining the modular effect specifications. When a concrete monad instance is needed, it can be easily obtained by interpretation using Isabelle's module system.

Second, as HOL can express the notion of a value-monomorphic monad, I have also formalised several monad transformers [20, 26] in HOL (§3). Thus, there is no need to define the monad and derive the reasoning principles for each combination of effects, as is current practice with value polymorphism. Instead, it suffices to formalise every effect only once as a transformer and combine them modularly.

Third, relations between different instances can be proven using the theory of representation independence (§4) as supported by Isabelle's Transfer package [14]. This makes it possible to switch in the middle of a bigger proof from a complicated monad to a simpler one.

In comparison to the conference version [23], this article additionally implements binary probabilistic choice from countable choice (§2.3) and presents monad transformers for non-determinism (§3.5) and a non-deterministic interpreter (§3.6).

2 Abstract Value-Monomorphic Monads in HOL

In this section, I specify value-monomorphic monads for several types of effects. A monadic interpreter for an arithmetic language will be used throughout as a running

[1] Available in the Archive of Formal Proofs at https://www.isa-afp.org/entries/Monomorphic_Monad.html.

example. The language, adapted from Nipkow and Klein [27], consists of integer constants, variables, addition, and division.

$\texttt{datatype}\ \nu\ \mathsf{exp} = \mathsf{Const\ int} \mid \mathsf{Var}\ \nu \mid (\nu\ \mathsf{exp}) \oplus (\nu\ \mathsf{exp}) \mid (\nu\ \mathsf{exp}) \oslash (\nu\ \mathsf{exp})$

I formalise the concept of a monad using Isabelle's module system of locales [2]. The locale monad below fixes the two monad operations return and bind (written infix as $\gg\!=$) and assumes that the monad laws hold. It will collect definitions of functions, which use the monad operations, and theorems about them, whose proofs can use the monad laws. Every locale also defines a predicate of the same name that collects all the assumptions; for example, monad $\mathsf{return}_{\mathsf{ident}}$ $\mathsf{bind}_{\mathsf{ident}}$ expresses that the two functions $\mathsf{return}_{\mathsf{ident}}$ and $\mathsf{bind}_{\mathsf{ident}}$ satisfy the monad laws. When a user interprets the locale with more concrete operations (e.g., $\mathsf{return}_{\mathsf{ident}}$ and $\mathsf{bind}_{\mathsf{ident}}$) and has discharged the assumptions for these operations, every definition and theorem inside the locale context is specialised to these operations. Although the type of values is a type variable α, α is fixed inside the locale. Instantiations may still replace α with any other HOL type. In other words, the locale monad formalises a *monomorphic* monad, but leaves the type of values unspecified. As usual, $m \gg m'$ abbreviates $m \gg\!= (\lambda\_.\ m')$.

$\texttt{locale}\ \mathsf{monad} = \texttt{fixes}\ \mathsf{return} :: \alpha \Rightarrow \mu\ \mathsf{and}\ \mathsf{bind} :: \mu \Rightarrow (\alpha \Rightarrow \mu) \Rightarrow \mu\ \ (\texttt{infixr}\ \gg\!=)$
$\quad\texttt{assumes}\ \textsc{bind-assoc}:\ (m \gg\!= f) \gg\!= g = m \gg\!= (\lambda x.\ f\ x \gg\!= g)$
$\quad\quad\mathsf{and}\ \textsc{return-bind}:\ \mathsf{return}\ x \gg\!= f = f\ x$
$\quad\quad\mathsf{and}\ \textsc{bind-return}:\ x \gg\!= \mathsf{return} = x$

Monads become useful only when effect-specific operations are available. In the remainder of this section, I formalise monadic operations for different types of effects and their properties. For each effect, I introduce a new locale in Isabelle that extends the locale monad, fixes the new operations, and specifies their properties. A locale extension inherits parameters and assumptions. This leads to a modular design: if several effects are needed, one merely combines the relevant locales in a multi-extension.

2.1 Failure and Exception

Failures are among the simplest effects and are widely used. A failure aborts the computation immediately. The locale monad-fail given below formalises the failure effect $\mathsf{fail} :: \mu$. It assumes that a failure propagates from the left hand side of bind, i.e., fail aborts a computation. In contrast, there is no assumption about how fail behaves on bind's right hand side. Otherwise, if monad-fail also assumed $m \gg\!= (\lambda\_.\ \mathsf{fail}) = \mathsf{fail}$, then fail would undo any effect of m. Although the standard implementation of failures using the option type satisfies this additional law, many other monad implementations do not, e.g., state-exception monads. Note that there is no need to delay the evaluation of fail in HOL because HOL has no execution semantics.

$\texttt{locale}\ \mathsf{monad\text{-}fail} = \mathsf{monad} + \texttt{fixes}\ \mathsf{fail} :: \mu$
$\quad\texttt{assumes}\ \textsc{fail-bind}:\ \mathsf{fail} \gg\!= f = \mathsf{fail}$

As a first example, I define the monadic interpreter $\mathsf{eval} :: (\nu \Rightarrow \mu) \Rightarrow \nu\ \mathsf{exp} \Rightarrow \mu$ for arithmetic expressions by primitive recursion using these abstract monad operations inside the locale monad-fail.[2] The first argument is an interpretation function $E :: \nu \Rightarrow \mu$ for the variables. The evaluation fails when a division by zero occurs.

$$
\begin{array}{l}
\texttt{primrec (in monad-fail) } \mathsf{eval} :: (\nu \Rightarrow \mu) \Rightarrow \nu\ \mathsf{exp} \Rightarrow \mu \texttt{ where} \\
\quad \mathsf{eval}\ E\ (\mathsf{Const}\ i) = \mathsf{return}\ i \\
\mid \mathsf{eval}\ E\ (\mathsf{Var}\ x) \quad = E\ x \\
\mid \mathsf{eval}\ E\ (e_1 \oplus e_2) = \mathsf{eval}\ E\ e_1 \gg\!= (\lambda i_1.\ \mathsf{eval}\ E\ e_2 \gg\!= (\lambda i_2.\ \mathsf{return}\ (i_1 + i_2))) \\
\mid \mathsf{eval}\ E\ (e_1 \oslash e_2) = \\
\qquad \mathsf{eval}\ E\ e_1 \gg\!= (\lambda i_1.\ \mathsf{eval}\ E\ e_2 \gg\!= (\lambda i_2.\ \mathsf{if}\ i_2 = 0\ \mathsf{then\ fail\ else\ return}\ (i_1\ \mathsf{div}\ i_2)))
\end{array}
$$

Note that evaluating a variable can have an effect μ, which is necessary to obtain a compositional interpreter. Let subst $:: (\nu \Rightarrow \nu'\ \mathsf{exp}) \Rightarrow \nu\ \mathsf{exp} \Rightarrow \nu'\ \mathsf{exp}$ be the substitution function for exp. That is, subst σ e replaces every Var x in e with $\sigma\ x$. Then, the following compositionality statement holds (proven by induction on e and term rewriting with the definitions), where function composition $\circ$ is defined as $(f \circ g)(x) = f\ (g\ x)$.

$$
\begin{array}{l}
\texttt{lemma } \textsc{compositionality}\text{:}\ \ \mathsf{eval}\ E\ (\mathsf{subst}\ \sigma\ e) = \mathsf{eval}\ (\mathsf{eval}\ E \circ \sigma)\ e \\
\quad \texttt{by } \textit{induction simp-all}
\end{array}
$$

I refer to failures as exceptions whenever there is an operator catch $:: \mu \Rightarrow \mu \Rightarrow \mu$ to handle them. Following Gibbons and Hinze [7], the locale monad-catch assumes that catch and fail form a monoid and that returns are not handled. It inherits FAIL-BIND and the monad laws by extending the locale monad-fail. No properties about the interaction between catch and bind are assumed because in general exception handling does not distribute over sequencing.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\texttt{locale } \mathsf{monad\text{-}catch} = \mathsf{monad\text{-}fail} + \texttt{fixes } \mathsf{catch} :: \mu \Rightarrow \mu \Rightarrow \mu} \\
\quad \texttt{assumes} & \textsc{fail-catch}\text{:} & \mathsf{catch\ fail}\ m = m \\
\quad\quad \texttt{and} & \textsc{catch-fail}\text{:} & \mathsf{catch}\ m\ \mathsf{fail} = m \\
\quad\quad \texttt{and} & \textsc{catch-catch}\text{:} & \mathsf{catch}\ (\mathsf{catch}\ m_1\ m_2)\ m_3 = \mathsf{catch}\ m_1\ (\mathsf{catch}\ m_2\ m_3) \\
\quad\quad \texttt{and} & \textsc{return-catch}\text{:} & \mathsf{catch}\ (\mathsf{return}\ x)\ m = \mathsf{return}\ x
\end{array}
$$

2.2 State

Some computations rely on a state that changes over time, e.g., counters, pseudo-random number generators, and destructive updates, where old versions are no longer needed. Such stateful computations use operations to read (get) and replace (put) the state of type σ. In a value-polymorphic setting, get $:: \sigma\ \tau$ and put $:: \sigma \Rightarrow \mathsf{unit}\ \tau$ are usually computations that return the state or () inhabiting the singleton type unit. Without value-polymorphism, these types cannot be formalised in the HOL setting because τ cannot be applied to different value types. Instead, my operations additionally take a continuation: get $:: (\sigma \Rightarrow \mu) \Rightarrow \mu$ and put $:: \sigma \Rightarrow \mu \Rightarrow \mu$. In a value-polymorphic setting, both signatures are equivalent. Passing the continuation return as in get return and $\lambda s.$ put s (return ()) yields the conventional operations. Conversely, my operations get f and put $s\ m$ can be implemented as get $\gg\!= f$ and put $s \gg m$ using conventional get and put. The locale monad-state collects the properties get and put must satisfy:

[2] Type variables that appear in the signature of locale parameters are fixed for the whole locale. In particular, the value type α cannot be instantiated inside the locale monad or its extension monad-fail. The interpreter eval, however, returns ints. For this reason, eval is defined in an extension of monad-fail that merely specialises α to int. For readability, I usually omit this detail in this article.

$\texttt{locale}$ monad-state $=$ monad $+$ $\texttt{fixes}$ get $:: (\sigma \Rightarrow \mu) \Rightarrow \mu$ and put $:: \sigma \Rightarrow \mu \Rightarrow \mu$

$\texttt{assumes}$ PUT-GET: put s (get f) $=$ put s $(f\ s)$

and GET-GET: get $(\lambda s.$ get $(f\ s)) =$ get $(\lambda s.\ f\ s\ s)$

and PUT-PUT: put s (put s' m) $=$ put s' m

and GET-PUT: get $(\lambda s.$ put $s\ m) = m$

and GET-CONST: get $(\lambda\_.\ m) = m$

and BIND-GET: get $f \gg\!= g =$ get $(\lambda s.\ f\ s \gg\!= g)$

and BIND-PUT: put $s\ m \gg\!= f =$ put $s\ (m \gg\!= f)$

The first four assumptions adapt Gibbons' and Hinze's axioms for the state operations [7] to the new signature. The fifth, GET-CONST, additionally specifies that get can be discarded if the state is not used. The last two assumptions, BIND-GET and BIND-PUT, demand that get and put distribute over bind. In the conventional value-polymorphic setting, where the continuations are applied using bind, these two are subsumed by the monad laws. In the remainder of this paper, get and put always take continuations.[3]

A state update function update can be implemented abstractly for all state monads. Like put, update takes a continuation m.

$\texttt{definition}$ (in monad-state) update $:: (\sigma \Rightarrow \sigma) \Rightarrow \mu \Rightarrow \mu$ $\texttt{where}$

update $f\ m =$ get $(\lambda s.$ put $(f\ s)\ m)$

The expected properties of update can be derived from monad-state's assumptions by term rewriting. For example,

$\texttt{lemma}$ UPDATE-ID: update id $m = m$

by (*simp add*: UPDATE-DEF GET-PUT)

$\texttt{lemma}$ UPDATE-UPDATE: update f (update $g\ m$) $=$ update $(g \circ f)\ m$

by (*simp add*: UPDATE-DEF PUT-GET PUT-PUT)

$\texttt{lemma}$ UPDATE-BIND: update $f\ m \gg\!= g =$ update $f\ (m \gg\!= g)$

by (*simp add*: UPDATE-DEF BIND-GET BIND-PUT)

As an example, I implement a memoisation operator memo using the state operations. To that end, the state must be refined to a lookup table, which I model as a map of type $\beta \rightharpoonup \alpha = \beta \Rightarrow \alpha$ option. The definition uses the function $\lambda t.\ t(x \mapsto y)$ that takes a map t and updates it to associate x with y, leaving the other associations as they are; formally, $t(x \mapsto y) = (\lambda x'.$ if $x = x'$ then Some y else $t\ x')$.

$\texttt{definition}$ (in monad-state) memo $:: (\beta \Rightarrow \mu) \Rightarrow \beta \Rightarrow \mu$ $\texttt{where}$

memo $f\ x =$ get $(\lambda \mathit{table}.$

case $\mathit{table}\ x$ of Some $y \Rightarrow$ return y

| None $\Rightarrow f\ x \gg\!= (\lambda y.$ update $(\lambda t.\ t(x \mapsto y))$ (return y)))

A memoisation operator should satisfy the following three properties. First, it evaluates the memoised function at most on the given argument, not on others. This can be expressed as a congruence rule, which holds by memo's definition:

[3] Continuation parameters like get's and put's make it possible to circumvent the restriction to monomorphic values. More generally, if we made every definition take a continuation, like in continuation-passing style, we would regain value polymorphism. In doing so, we would however lose that sequencing of computations and control flow is captured by a small number of (primitive) operations. Instead, every definition could implement arbitrary control flow, like in a continuation monad. Thus, we would need a lot more lemmas to reason about sequencing. In a commutative monad, e.g., one commutativity lemma would be needed for *every* pair of definitions. In contrast, my approach needs just *one* assumption COMM to express commutativity of sequencing (§2.5), and a few assumptions about the primitive operations, e.g., SAMPLE-COMM (§2.3).

lemma MEMO-CONG: $f\ x = g\ x \longrightarrow \mathsf{memo}\ f\ x = \mathsf{memo}\ g\ x$

Second, memoisation should be idempotent, i.e., if a function is already being memoised, then there is no point in memoising it once more.

lemma MEMO-IDEM: $\mathsf{memo}\ (\mathsf{memo}\ f)\ x = \mathsf{memo}\ f\ x$

The mechanised proof of MEMO-IDEM in Isabelle needs only two steps, which are justified by term rewriting with the properties of the monad operations and the case operator. Every assumption about get and put except GET-PUT is needed. Appendix A contains a step-by-step proof that illustrates reasoning with the algebraic monad properties.

Third, the memoisation operator should indeed evaluate f on x at most once. As $\mathsf{memo}\ f\ x$ memoises only the result of $f\ x$, but not the effect of evaluating $f\ x$, the next lemma captures this correctness property. Its proof is similar to MEMO-IDEM's.

lemma CORRECT: $\mathsf{memo}\ f\ x \gg\!= (\lambda a.\ \mathsf{memo}\ f\ x \gg\!= g\ a) = \mathsf{memo}\ f\ x \gg\!= (\lambda a. g\ a\ a)$

2.3 Probabilistic Choice

Randomised computations are built from an operation ¢ for probabilistic choice. The probabilities are specified using probability mass functions (type π pmf) [11], i.e., discrete probability distributions. Binary probabilistic choice, which is often used in the literature [6, 7, 30], is less general as it leads to finite distributions. Continuous distributions would work, too, but they would require measurability conditions everywhere.

Like the state operations, $¢ :: \pi\ \mathsf{pmf} \Rightarrow (\pi \Rightarrow \mu) \Rightarrow \mu$ takes a continuation to separate the type of probabilistic choices π from the type of values. The locale monad-prob assumes the following properties:

- sampling from the one-point distribution $\mathsf{dirac}\ x$ has no effect (SAMPLE-DIRAC),
- sequencing $\mathsf{bind}_{\mathsf{pmf}}$ in the probability monad yields sequencing (SAMPLE-BIND),
- sampling can be discarded if the result is unused (SAMPLE-CONST),
- sampling from independent distributions commutes (SAMPLE-COMM, independence is expressed by p and q not taking y and x as an argument, respectively.)
- sampling is relationally parametric in the choices (SAMPLE-PARAM), and
- sampling distributes over both sides of bind (BIND-SAMPLE$_1$, BIND-SAMPLE$_2$).

locale monad-prob = monad + fixes $¢ :: \pi\ \mathsf{pmf} \Rightarrow (\pi \Rightarrow \mu) \Rightarrow \mu$
  assumes SAMPLE-DIRAC: $¢\ (\mathsf{dirac}\ x)\ f = f\ x$
    and SAMPLE-BIND: $¢\ (\mathsf{bind}_{\mathsf{pmf}}\ p\ f)\ g = ¢\ p\ (\lambda x.\ ¢\ (f\ x)\ g)$
    and SAMPLE-CONST: $¢\ p\ (\lambda\_.\ m) = m$
    and SAMPLE-COMM: $¢\ p\ (\lambda x.\ ¢\ q\ (f\ x)) = ¢\ q\ (\lambda y.\ ¢\ p\ (\lambda x.\ f\ x\ y))$
    and SAMPLE-PARAM: $\mathsf{bi\text{-}unique}\ R \longrightarrow (¢, ¢) \in \mathsf{rel}_{\mathsf{pmf}}\ R \Mapsto (R \Mapsto (=)) \Mapsto (=)$
    and BIND-SAMPLE$_1$: $¢\ p\ f \gg\!= g = ¢\ p\ (\lambda x.\ f\ x \gg\!= g)$
    and BIND-SAMPLE$_2$: $m \gg\!= (\lambda x.\ ¢\ p\ (f\ x)) = ¢\ p\ (\lambda y.\ m \gg\!= (\lambda x.\ f\ x\ y))$

The assumption SAMPLE-PARAM ensures that ¢ does not look at the identity of the choices. This is expressed as a Reynolds-style parametricity condition [31] where

- R is a relation between the choices, where the condition bi-unique expresses that R relates each choice with at most one choice, i.e., R does not identify choices;[4]

[4] In my monad implementations in §3, sampling is relationally parametric for arbitrary relations R, so I could drop the restriction bi-unique R. However, this would unnecessarily exclude some other implementations as all my abstract proofs so far used only bi-unique relations.

– the relator $\mathsf{rel_{pmf}}$ lifts a relation on elementary events to probability distributions: $(p,q) \in \mathsf{rel_{pmf}}\ A$ iff $\mathcal{P}[p \in X] \le \mathcal{P}[q \in \{y \mid \exists x \in X.\ (x,y) \in A\}]$ for all X, where $\mathcal{P}[p \in X]$ denotes the probability of the event X under the distribution p;
– the right-associative function relator $A \Mapsto B$ relates two functions f and g iff $(x,y) \in A$ implies $(f(x), g(y)) \in B$ for all x and y; and
– $(=)$ denotes the identity relation.

For example, consider two biased coins p_1 and p_2 that show heads with probabilities r and $1-r$, respectively. Then, it should not matter whether we flip p_1 or p_2 provided that we switch the actions for heads and tails. Formally, $¢\ p_1\ f = ¢\ p_2\ g$ if $f\ \mathsf{heads} = g\ \mathsf{tails}$ and $f\ \mathsf{tails} = g\ \mathsf{heads}$. This identity follows from SAMPLE-PARAM using $R = \{(\mathsf{heads}, \mathsf{tails}), (\mathsf{tails}, \mathsf{heads})\}$.

Parametricity in particular ensures that $¢\ p\ f$ calls the continuation f only on values in p's support[5] $\mathsf{supp}\ p$ (take $R = \{(x,x) \mid x \in \mathsf{supp}\ p\}$ in SAMPLE-PARAM):[6]

`lemma` (`in` monad-prob) SAMPLE-CONG: $(\forall x \in \mathsf{supp}\ p.\ f\ x = g\ x) \longrightarrow ¢\ p\ f = ¢\ p\ g$

Binary probabilistic choice $m_1 \lhd r \rhd m_2$ can be defined in terms of ¢. It behaves as m_1 with probability r and as m_2 with probability $1-r$. For it to be well-behaved, we must require that the type π of choices contains at least three choices, say ①, ②, and ③. Let $\mathsf{flip}\ r :: \pi\ \mathsf{pmf}$ be the distribution that assigns probability r to ① and $1-r$ to ②. (The third choice ③ is needed to prove the associativity law below.)

`definition` $\_ \lhd \_ \rhd \_ :: \mu \Rightarrow \mathsf{real} \Rightarrow \mu \Rightarrow \mu$ `where`
$m_1 \lhd r \rhd m_2 = ¢\ (\mathsf{flip}\ r)\ (\lambda x.\ \mathsf{if}\ x = ①\ \mathsf{then}\ m_1\ \mathsf{else}\ m_2)$

From the monad-prob assumptions, I can derive Gibbons and Hinze's specification [7]. All assumptions are used except SAMPLE-COMM. Associativity crucially relies on SAMPLE-PARAM and the existence of a third choice ③ as we must distribute the probability over three computations, not just the two of the inner choice operator.

`lemma` CHOOSE-0: $m \lhd 0 \rhd m' = m'$

`lemma` CHOOSE-1: $m \lhd 1 \rhd m' = m$

`lemma` CHOOSE-IDEM: $m \lhd r \rhd m = m$

`lemma` CHOOSE-COMMUTE: $m \lhd 1-r \rhd m' = m' \lhd r \rhd m$

`lemma` CHOOSE-BIND: $(m \lhd r \rhd m') \gg\!= f = (m \gg\!= f) \lhd r \rhd (m' \gg\!= f)$

`lemma` BIND-CHOOSE: $m \gg\!= (\lambda x.\ f\ x \lhd r \rhd g\ x) = (m \gg\!= f) \lhd r \rhd (m \gg\!= g)$

`lemma` CHOOSE-ASSOC: $m_1 \lhd p \rhd (m_2 \lhd q \rhd m_3) = (m_1 \lhd r \rhd m_2) \lhd s \rhd m_3$
`if` $p = r * s$ `and` $1 - s = (1-p) * (1-q)$

2.4 Combining Abstract Monads

Formalising monads in this abstract way has the advantage that the different effects can be easily combined. In the running example, suppose that the variables represent independent random variables. Then, expressions are probabilistic computations and evaluation computes the joint probability distribution. For example, if x_1 and

[5] The support of p is the set of elementary events with positive probability: $x \in \mathsf{supp}\ p$ iff $\mathcal{P}[p \in \{x\}] > 0$.
[6] The conference paper [23] demanded SAMPLE-CONG instead of SAMPLE-PARAM. Parametricity is a better condition as it allows us to rename choices like in the biased coin flip example above.

x_2 represent coin flips with 1 representing heads and 0 tails, then $\mathsf{Var}\ x_1 \oplus \mathsf{Var}\ x_2$ represents the probability distribution of the number of heads.

Here is a first attempt. Let $X :: \nu \Rightarrow \mathsf{int\ pmf}$ specify the distribution $X\ x$ for each random variable x. Combining the locales for failures and probabilistic choice, we let the variable environment do the sampling, where $\mathsf{sample\text{-}var}\ X\ x = \text{¢}\ (X\ x)\ \mathsf{return}$:

`locale` monad-fail-prob = monad-fail + monad-prob

`definition` (`in` monad-fail-prob) wrong :: $(\nu \Rightarrow \mathsf{int\ pmf}) \Rightarrow \nu\ \mathsf{exp} \Rightarrow \mu$ `where`
wrong $X\ e$ = eval (sample-var X) e

As the name suggests, wrong does not achieve what was intended. If a variable occurs multiple times in e, say $e = \mathsf{Var}\ x \oplus \mathsf{Var}\ x$, then wrong $X\ e$ samples x afresh for each occurrence. So, if $X\ x = \mathsf{uniform}\ \{0,1\}$, i.e., x is a coin flip, wrong $X\ e$ computes the probability distribution given by $0 \mapsto 1/4, 1 \mapsto 1/2, 2 \mapsto 1/4$ instead of $0 \mapsto 1/2, 2 \mapsto 1/2$. Clearly, every variable should be sampled at most once. Memoising the variable evaluation achieves that. So, we additionally need state operations.

`locale` monad-fail-prob-state = monad-fail-prob + monad-state +
`assumes` SAMPLE-GET: $\text{¢}\ p\ (\lambda x.\ \mathsf{get}\,(f\ x)) = \mathsf{get}\,(\lambda s.\ \text{¢}\ p\ (\lambda x.\ f\ x\ s))$

`definition` (`in` monad-fail-prob-state) lazy :: $(\nu \Rightarrow \mathsf{int\ pmf}) \Rightarrow \nu\ \mathsf{exp} \Rightarrow \mu$ `where`
lazy $X\ e$ = eval (memo (sample-var X)) e

The interpreter lazy samples a variable only when needed. For example, in $e_0 = (\mathsf{Const}\ 1 \oslash \mathsf{Const}\ 0) \oplus \mathsf{Var}\ x_0$, the division by zero makes the evaluation fail before x_0 is sampled.

The locale monad-fail-prob-state adds an assumption that ¢ distributes over get. Such distributivity assumptions are typically needed because of the continuation parameters, which break the separation between effects and sequencing. Their format is as follows: If two operations f_1 and f_2 with continuations do not interact, then we assume $f_1\ (\lambda x.\ f_2\ (g\ x)) = f_2\ (\lambda y.\ f_1\ (\lambda x.\ g\ x\ y))$. Sometimes, such assumptions follow from existing assumptions. For example, SAMPLE-PUT follows from BIND-SAMPLE$_2$ and $\mathsf{put}\ s\ m = \mathsf{put}\ s\ (\mathsf{return}\ x) \gg m$ for all x. A similar law holds for update.

`lemma` SAMPLE-PUT: $\text{¢}\ p\ (\lambda x.\ \mathsf{put}\ s\ (f\ x)) = \mathsf{put}\ s\ (\text{¢}\ p\ f)$

In contrast, SAMPLE-GET does not follow from the other assumptions due to the restriction to monomorphic values. The state of type σ, which get passes to its continuation, may carry more information than a value can hold. Indeed, in the case of lazy, the type int of values is countable, but the state type $\nu \rightharpoonup \mathsf{int}$ is not if the type of variables is infinite. As put passes no information to its continuation, put's continuation can be pushed into bind as shown above. Still, put needs its continuation; otherwise, it would have to create a return value out of nothing, which would cause problems later (§4). Moreover, there is no need to explicitly specify how fail interacts with get and ¢ as $\mathsf{get}\ (\lambda\_.\ \mathsf{fail}) = \mathsf{fail}$ and $\text{¢}\ p\ (\lambda\_.\ \mathsf{fail}) = \mathsf{fail}$ are special cases of GET-CONST and SAMPLE-CONST.

Instead of lazy sampling, we can also sample all variables eagerly. Let vars e return the (finite) set of variables in e. Then, the interpreter eager with eager sampling is defined as follows (all three definitions live in the locale monad-fail-prob-state):

`definition` sample-vars :: $(\nu \Rightarrow \mathsf{int\ pmf}) \Rightarrow \nu\ \mathsf{set} \Rightarrow \mu \Rightarrow \mu$ `where`
sample-vars $X\ A\ m$ = fold $(\lambda x\ m.\ \mathsf{memo}\ (\mathsf{sample\text{-}var}\ X)\ x \gg m)\ m\ A$

`definition` lookup :: $\nu \Rightarrow \mu$ `where`
lookup x = get $(\lambda s.\ \mathsf{case}\ s\ x\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{fail} \mid \mathsf{Some}\ i \Rightarrow \mathsf{return}\ i)$

$\texttt{definition}$ $\mathsf{eager} :: (\nu \Rightarrow \mathsf{int\ pmf}) \Rightarrow \nu\ \mathsf{exp} \Rightarrow \mu$ $\texttt{where}$

$\quad \mathsf{eager}\ X\ e = \mathsf{sample\text{-}vars}\ X\ (\mathsf{vars}\ e)\ (\mathsf{eval\ lookup}\ e)$

where fold is the fold operator for finite sets [28]. The operator fold h requires that the folding function h is left-commutative, i.e., $h\ x\ (h\ y\ z) = h\ y\ (h\ x\ z)$ for all x, y, and z. In our case, $h = \lambda x\ m.\ \mathsf{memo}\ (\mathsf{sample\text{-}var}\ X)\ x \gg m$ is left-commutative by the following lemma about memo whose assumptions hold for $f = \mathsf{sample\text{-}var}\ X$ thanks to RETURN-BIND, BIND-SAMPLE$_1$, BIND-SAMPLE$_2$, and SAMPLE-GET. Moreover, by CORRECT, it is also idempotent, i.e., $h\ x \circ h\ x = h\ x$.

$\texttt{lemma}$ MEMO-COMMUTE:

$$
\begin{aligned}
&\quad (\forall m\ x\ g.\ m \gg\!= (\lambda a.\ f\ x \gg\!= g\ a) = f\ x \gg\!= (\lambda b.\ m \gg\!= (\lambda a.\ g\ a\ b)))\\
&\longrightarrow (\forall x\ g.\ \mathsf{get}\ (\lambda s.\ f\ x \gg\!= g\ s) = f\ x \gg\!= (\lambda a.\ \mathsf{get}\ (\lambda s.\ g\ s\ a)))\\
&\longrightarrow \mathsf{memo}\ f\ x \gg\!= (\lambda a.\ \mathsf{memo}\ f\ y \gg\!= (\lambda b.\ g\ a\ b)) =\\
&\quad \mathsf{memo}\ f\ y \gg\!= (\lambda b.\ \mathsf{memo}\ f\ x \gg\!= (\lambda a.\ g\ a\ b))
\end{aligned}
$$

This lemma and CORRECT illustrate the typical form of monadic statements. The assumptions and conclusions take a continuation g for the remainder of the program. This way, the statements are easier to apply because they are in normal form with respect to BIND-ASSOC. This observation also holds in a value-polymorphic setting.

Now, the question is whether eager and lazy sampling are equivalent. In general, the answer is no. For example, for e_0 from above, $\mathsf{eager}\ X\ e_0$ samples and memoises the variable x_0, but $\mathsf{lazy}\ X\ e_0$ does not. Thus, there are contexts that distinguish the two. If we extend monad-fail-prob-state with exception handling from monad-catch such that get and put never fail,

CATCH-GET: $\mathsf{catch}\ (\mathsf{get}\ f)\ m_2 = \mathsf{get}\ (\lambda s.\ \mathsf{catch}\ (f\ s)\ m_2)$

CATCH-PUT: $\mathsf{catch}\ (\mathsf{put}\ s\ m)\ m_2 = \mathsf{put}\ s\ (\mathsf{catch}\ m\ m_2)$

then the two can be distinguished:

$$
\begin{aligned}
\mathsf{catch}\ (\mathsf{lazy}\ X\ e_0)\ (\mathsf{lookup}\ x_0) &= \mathsf{fail}\\
\mathsf{catch}\ (\mathsf{eager}\ X\ e_0)\ (\mathsf{lookup}\ x_0) &= \mathsf{memo}\ (\mathsf{sample\text{-}var}\ X)\ x_0
\end{aligned}
$$

In contrast, if we assume that failures erase state updates, then the two *are* equivalent:

$\texttt{theorem}$ LAZY-EAGER: $(\forall s.\ \mathsf{put}\ s\ \mathsf{fail} = \mathsf{fail}) \longrightarrow \mathsf{lazy}\ X\ e = \mathsf{eager}\ X\ e$

Proof The proof consists of three steps proven by induction on e. First, by idempotence and left-commutativity, $\mathsf{sample\text{-}vars}\ X\ V$ commutes with $\mathsf{lazy}\ X\ e$ for any finite V:

$$\forall g.\ \mathsf{sample\text{-}vars}\ X\ V\ (\mathsf{lazy}\ X\ e \gg\!= g) = \mathsf{lazy}\ X\ e \gg\!= (\lambda i.\ \mathsf{sample\text{-}vars}\ X\ V\ (g\ i)) \quad (1)$$

Here, $\mathsf{put}\ s\ \mathsf{fail} = \mathsf{fail}$ ensures that all state updates are lost if a division by zero occurs. The next two steps will use (1) in the inductive cases for $\oplus$ and $\oslash$ to bring together the sampling of the variables and the evaluation of the subexpressions. Second,

$$\mathsf{lazy}\ X\ e \gg\!= g = \mathsf{sample\text{-}vars}\ X\ (\mathsf{vars}\ e)\ (\mathsf{lazy}\ X\ e \gg\!= g) \quad (2)$$

shows that the sampling can be done first, which holds by CORRECT. Finally,

$$\mathsf{sample\text{-}vars}\ X\ V\ (\mathsf{lazy}\ X\ e \gg\!= g) = \mathsf{sample\text{-}vars}\ X\ V\ (\mathsf{eval\ lookup}\ e \gg\!= g) \quad (3)$$

holds for any finite set V with $\mathsf{vars}\ e \subseteq V$. Here, $\mathsf{Var}\ x$ is the interesting case, which follows from $\forall g.\ \mathsf{memo}\ f\ x \gg\!= (\lambda i.\ \mathsf{lookup}\ x \gg\!= g\ i) = \mathsf{memo}\ f\ x \gg\!= (\lambda i.\ g\ i\ i)$ and CORRECT. Taking $V = \mathsf{vars}\ e$ and $g = \mathsf{return}$, (2) and (3) prove the theorem. □

In §3.6, I show that some monads satisfy LAZY-EAGER's assumption, but not all.

2.5 Abstract Monad Properties

Some monad transformer implementations require that the transformed monad satisfies additional properties. I consider three properties, which I formalise as locales:

- A monad is *commutative* if independent computations can be reordered.
- A monad is *discardable* if we may drop a computation whose result is not used.
- A monad is *duplicable* if a computation may be duplicated.

$\texttt{locale}$ $\textsf{monad-comm} = \textsf{monad} +$
  $\texttt{assumes}$ COMM: $m_1 \gg\!= (\lambda x.\ m_2 \gg\!= f\ x) = m_2 \gg\!= (\lambda y.\ m_1 \gg\!= (\lambda x.\ f\ x\ y))$

$\texttt{locale}$ $\textsf{monad-discard} = \textsf{monad} +$
  $\texttt{assumes}$ DISCARD: $m_1 \gg m_2 = m_2$

$\texttt{locale}$ $\textsf{monad-dup} = \textsf{monad} +$
  $\texttt{assumes}$ DUP: $m \gg\!= (\lambda x.\ m \gg\!= f\ x) = m \gg\!= (\lambda x.\ f\ x\ x)$

2.6 Further Abstract Monads

Apart from exceptions, state, and probabilistic choice, I have formalised specifications for the following effects. I only present them to the level of detail needed for understanding the remainder of this paper.

Non-determinism. Non-determinism is often used in specification where implementation choices are unspecified; implementations can then refine the non-determinism [1,19]. Backtracking can also be implemented elegantly using non-deterministic choice and failure [33]. I specify two choice operators: binary $\textsf{alt} :: \mu \Rightarrow \mu \Rightarrow \mu$ and countable $\textsf{altc} :: \chi\ \textsf{cset} \Rightarrow (\chi \Rightarrow \mu) \Rightarrow \mu$ where the type χ $\textsf{cset}$ consists of all countable sets over χ. Unbounded choice would be similar to countable choice. Their specification is similar to probabilities, but I demand less to allow more implementations in §3.5. The laws for $\textsf{alt}$ are taken from Gibbons and Hinze [7].

$\texttt{locale}$ $\textsf{monad-alt} = \textsf{monad return bind} + \texttt{fixes}$ $\textsf{alt} :: \mu \Rightarrow \mu \Rightarrow \mu$
  $\texttt{assumes}$ ALT-ASSOC: $\textsf{alt}\ (\textsf{alt}\ m_1\ m_2)\ m_3 = \textsf{alt}\ m_1\ (\textsf{alt}\ m_2\ m3)$
    $\texttt{and}$ ALT-BIND: $\textsf{alt}\ m\ m' \gg\!= f = \textsf{alt}\ (m \gg\!= f)\ (m' \gg\!= f)$

$\texttt{locale}$ $\textsf{monad-altc} = \textsf{monad return bind} + \texttt{fixes}$ $\textsf{altc} :: \chi\ \textsf{cset} \Rightarrow (\alpha \Rightarrow \mu) \Rightarrow \mu$
  $\texttt{assumes}$ ALTC-BIND: $\textsf{altc}\ C\ f \gg\!= g = \textsf{altc} C\ (\lambda c.\ f\ c \gg\!= g)$
    $\texttt{and}$ ALTC-SINGLE: $\textsf{altc}\ \{x\}\ f = f\ x$
    $\texttt{and}$ ALTC-UNION: $\textsf{altc}\ (\bigcup_{c \in C}\ g\ c)\ f = \textsf{altc}\ C\ (\lambda c.\ \textsf{altc}\ (f\ c)\ g)$
    $\texttt{and}$ ALT-PARAM: $\textsf{bi-unique}\ R \longrightarrow (\textsf{altc}, \textsf{altc}) \in \textsf{rel}_{\textsf{cset}}\ R \Mapsto (R \Mapsto (=)) \Mapsto (=)$

Reader and writer monads. The reader monad makes it possible to pass immutable data to (many) functions without threading the parameter through. For example, a security parameter in cryptography is typically hidden in pen-and-paper notation and a reader monad achieves the same in a formalisation. Environments, e.g., variable bindings, are also good candidates for a reader monad. The operation $\textsf{ask} :: (\rho \Rightarrow \mu) \Rightarrow \mu$ retrieves the data of type ρ.

The writer monad allows programs to output data in several steps using $\textsf{tell} :: \omega \Rightarrow \mu \Rightarrow \mu$. Unlike updates in the state monad, outputs cannot be made undone. It is in particular suitable for logging.

Resumption. Resumptions provide a semantic domain for reactive and concurrent programs [9, 29]. The primitive operation $\mathsf{pause} :: \mathrm{o} \Rightarrow (\iota \Rightarrow \mu) \Rightarrow \mu$ interrupts a computation to output a value of type o and waits for some input of type ι before the computation resumes.

3 Implementations of Monads and Monad Transformers

In the previous section, I specified the properties of monadic operations abstractly. Now, I provide monad implementations that satisfy these specifications. Some effects are implemented as monad transformers [20, 26], which allow me to compose implementations of different effects almost as modularly as the locales specifying them abstractly. In particular, I analyse whether the transformers preserve the specifications of the other effects. All implementations are polymorphic in the values such that they can be used with any value type, although by the value-monomorphism restriction, each usage must individually commit to one value type.

3.1 The Identity Monad

The simplest monad implementation in my library is the identity monad ident, which models the absence of all effects. It is not really useful in itself, but will be an important building block when combining monads using transformers. The datatype α ident is a copy of α with constructor Ident and selector run-ident. To distinguish the abstract monad operations from their implementations, I subscript the latter with the implementation type. The lemma states that $\mathsf{return}_{\mathsf{ident}}$ and $\mathsf{bind}_{\mathsf{ident}}$ satisfy the assumption of the locale monad. Moreover, the identity monad is commutative, discardable, and duplicable.

`datatype` α ident = Ident (run-ident: α)
`definition` $\mathsf{return}_{\mathsf{ident}} :: \alpha \Rightarrow \alpha$ ident `where` $\mathsf{return}_{\mathsf{ident}}$ = Ident
`definition` $\mathsf{bind}_{\mathsf{ident}} :: \alpha$ ident $\Rightarrow (\alpha \Rightarrow \alpha$ ident$) \Rightarrow \alpha$ ident `where`
  $m \gg\!=_{\mathsf{ident}} f = f$ (run-ident m)

`lemma` monad $\mathsf{return}_{\mathsf{ident}}$ $\mathsf{bind}_{\mathsf{ident}}$

3.2 The Probability Monad

The probability monad α prob is another basic building block. I use discrete probability distributions [11] and Giry's probability monad operations dirac and $\mathsf{bind}_{\mathsf{pmf}}$, which I already used in the abstract specification in §2.3. Then, probabilistic choice $¢_{\mathsf{prob}}$ is just monadic sequencing on α pmf. The probability monad is commutative and discardable.

`type-synonym` α prob = α pmf
`definition` $\mathsf{return}_{\mathsf{prob}} :: \alpha \Rightarrow \alpha$ prob `where` $\mathsf{return}_{\mathsf{prob}}$ = dirac
`definition` $\mathsf{bind}_{\mathsf{prob}} :: \alpha$ prob $\Rightarrow (\alpha \Rightarrow \alpha$ prob$) \Rightarrow \alpha$ prob `where` $\mathsf{bind}_{\mathsf{prob}} = \mathsf{bind}_{\mathsf{pmf}}$
`definition` $¢_{\mathsf{prob}} :: \pi$ pmf $\Rightarrow (\pi \Rightarrow \alpha$ prob$) \Rightarrow \alpha$ prob `where` $¢_{\mathsf{prob}} = \mathsf{bind}_{\mathsf{pmf}}$

`lemma` monad-prob $\mathsf{return}_{\mathsf{prob}}$ $\mathsf{bind}_{\mathsf{prob}}$ $¢_{\mathsf{prob}}$

Failures and exception handling are implemented as a monad transformer. Thus, these effects can be added to any monad τ. In the value-polymorphic setting, the failure monad transformer takes a monad τ and defines a type constructor failT such that β failT is isomorphic to (β option) τ. That is, the transformer specialises the value type α of the inner monad to β option. In the value-monomorphic setting, the type variable μ represents the application of τ to the value type, i.e., β option. So, μ failT is just a copy of μ:

$\texttt{datatype}\ \mu\ \mathsf{failT} = \mathsf{FailT}\ (\mathsf{run\text{-}fail} : \mu)$

As failT's operations depend on the inner monad, I fix abstract operations return and bind in an unnamed context and define failT's operations in terms of them. The line on the left indicates the scope of the context. At the end, which is marked by ∟, the fixed operations become additional arguments of the defined functions. Values in the inner monad now have type α option. The definitions themselves are standard [26].

$\texttt{context fixes}\ \mathsf{return} :: \alpha\ \mathsf{option} \Rightarrow \mu\ \texttt{and}\ \mathsf{bind} :: \mu \Rightarrow (\alpha\ \mathsf{option} \Rightarrow \mu) \Rightarrow \mu$

$\texttt{definition}\ \mathsf{return}_{\mathsf{failT}} :: \alpha \Rightarrow \mu\ \mathsf{failT}\ \texttt{where}$
$\quad \mathsf{return}_{\mathsf{failT}}\ x = \mathsf{FailT}\ (\mathsf{return}\ (\mathsf{Some}\ x))$

$\texttt{definition}\ \mathsf{bind}_{\mathsf{failT}} :: \mu\ \mathsf{failT} \Rightarrow (\alpha \Rightarrow \mu\ \mathsf{failT}) \Rightarrow \mu\ \mathsf{failT}\ \texttt{where}$
$\quad m \ggg_{\mathsf{failT}} f = \mathsf{FailT}\ (\mathsf{run\text{-}fail}\ m \ggg$
$\qquad (\lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{return}\ \mathsf{None} \mid \mathsf{Some}\ y \Rightarrow \mathsf{run\text{-}fail}\ (f\ y)))$

$\texttt{definition}\ \mathsf{fail}_{\mathsf{failT}} :: \mu\ \mathsf{failT}\ \texttt{where}\ \mathsf{fail}_{\mathsf{failT}} = \mathsf{FailT}\ (\mathsf{return}\ \mathsf{None})$

$\texttt{definition}\ \mathsf{catch}_{\mathsf{failT}} :: \mu\ \mathsf{failT} \Rightarrow \mu\ \mathsf{failT} \Rightarrow \mu\ \mathsf{failT}\ \texttt{where}$
$\quad \mathsf{catch}_{\mathsf{failT}}\ m_1\ m_2 = \mathsf{FailT}\ (\mathsf{run\text{-}fail}\ m_1 \ggg$
$\qquad (\lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{None} \Rightarrow \mathsf{run\text{-}fail}\ m_2 \mid \mathsf{Some}\ \_ \Rightarrow \mathsf{return}\ x))$

If return and bind form a monad, so do $\mathsf{return}_{\mathsf{failT}}$ and $\mathsf{bind}_{\mathsf{failT}}$, and $\mathsf{fail}_{\mathsf{failT}}$ and $\mathsf{catch}_{\mathsf{failT}}$ satisfy the effect specification from §2.1, too. The next lemma expresses this.

$\texttt{lemma}\ \mathsf{monad\text{-}catch}\ \mathsf{return}_{\mathsf{failT}}\ \mathsf{bind}_{\mathsf{failT}}\ \mathsf{fail}_{\mathsf{failT}}\ \mathsf{catch}_{\mathsf{failT}}$
$\quad \texttt{if}\ \mathsf{monad}\ \mathsf{return}\ \mathsf{bind}$

The monad μ failT is

- commutative if the inner monad μ is commutative and discardable;
- duplicable if μ is duplicable and discardable; and
- not discardable, e.g., $\mathsf{fail}_{\mathsf{failT}} \gg \mathsf{return}_{\mathsf{failT}}\ x = \mathsf{fail}_{\mathsf{failT}} \neq \mathsf{return}_{\mathsf{failT}}\ x$.

Clearly, I want to keep using the existing effects of the inner monad. So, I must lift their operations to failT and prove that their specifications are preserved. The lifting is not hard; the continuations of the operations are transformed in the same way as $\mathsf{bind}_{\mathsf{failT}}$ transforms its continuation. Here, I only show how to lift the state operations, where the locale $\mathsf{monad\text{-}catch\text{-}state}$ extends $\mathsf{monad\text{-}catch}$ and $\mathsf{monad\text{-}state}$ with CATCH-GET and CATCH-PUT. Moreover, failT also lifts ¢, alt, altc, ask, tell, and pause, preserving their specifications.

```
context fixes get :: (σ ⇒ μ) ⇒ μ and put :: σ ⇒ μ ⇒ μ
  definition get_failT :: (σ ⇒ μ failT) ⇒ μ failT where
    get_failT f = FailT (get (λs. run-fail (f s)))
  definition put_failT :: σ ⇒ μ failT ⇒ μ failT where
    put_failT s m = FailT (put s (run-fail m))
  lemma monad-catch-state return_failT bind_failT fail_failT catch_failT get_failT put_failT
    if monad-state return bind get put
```

From now on, as the context scope has ended, $\mathsf{return}_{\mathsf{failT}}$ and $\mathsf{bind}_{\mathsf{failT}}$ take the inner monad's operations return and bind as additional arguments. For example, I obtain a plain failure monad by applying failT to ident. Interpreting the locale monad-fail for $\mathsf{return}_{\mathrm{F}} = \mathsf{return}_{\mathsf{failT}}\ \mathsf{return}_{\mathsf{ident}}$ and $\mathsf{bind}_{\mathrm{F}} = \mathsf{bind}_{\mathsf{failT}}\ \mathsf{return}_{\mathsf{ident}}\ \mathsf{bind}_{\mathsf{ident}}$ and $\mathsf{fail}_{\mathrm{F}} = \mathsf{fail}_{\mathsf{failT}}\ \mathsf{return}_{\mathsf{ident}}$ yields an executable version of the interpreter eval from §2.1, which I refer to as $\mathsf{eval}_{\mathrm{F}}$. Then, Isabelle's code generator and simplifier both evaluate

$$\mathsf{eval}_{\mathrm{F}}\ (\lambda x.\ \mathsf{return}_{\mathrm{F}}\ (((\lambda\_.\ 0)(x_0 := 5))\ x))\ (\mathsf{Var}\ x_0 \oplus \mathsf{Const}\ 7)$$

to FailT (Ident (Some 12)). Given some variable environment $Y :: \nu \Rightarrow \mathsf{int}$,[7] I obtain a textbook-style interpreter [27, §3.1.2] as run-ident (run-fail ($\mathsf{eval}_{\mathrm{F}}$ ($\mathsf{return}_{\mathrm{F}} \circ Y$) e)).

3.4 The State Monad Transformer

The state monad transformer adds the effects of a state monad to some inner monad. The formalisation follows the same ideas as for failT, so I only mention the important points. The state monad transformer transforms a monad $\alpha\ \tau$ into the type $\sigma \Rightarrow (\alpha \times \sigma)\ \tau$ where σ is the type of states. So, in HOL, the type of values of the inner monad becomes $\alpha \times \sigma$ and μ represents $(\alpha \times \sigma)\ \tau$.

```
datatype (σ, μ) stateT = StateT (run-state: σ ⇒ μ)
```

Like for failT, the state monad operations $\mathsf{return}_{\mathsf{stateT}}$ and $\mathsf{bind}_{\mathsf{state}}$ depend on inner monad operations return and bind. With $\mathsf{get}_{\mathsf{stateT}}$ and $\mathsf{put}_{\mathsf{stateT}}$ defined in the obvious way, the transformer satisfies the specification monad-state for state monads.

```
context fixes return :: α × σ ⇒ μ and bind :: μ ⇒ (α × σ ⇒ μ) ⇒ μ
  definition return_stateT :: α ⇒ (σ, μ) stateT where
    return_stateT x = StateT (λs. return (x, s))
  definition bind_stateT :: (σ, μ) stateT ⇒ (α ⇒ (σ, μ) stateT) ⇒ (σ, μ) stateT where
    m >>=_stateT f = StateT (λs. run-state f s >>= (λ(x, s'). run-state (f x) s'))
  definition get_stateT :: (σ ⇒ (σ, μ) stateT) ⇒ (σ, μ) stateT where
    get_stateT f = StateT (λs. run-state (f s) s)
  definition put_stateT :: σ ⇒ (σ, μ) stateT ⇒ (σ, μ) stateT where
    put_stateT s m = StateT (λ_. run-state m s)
  lemma monad-state return_stateT bind_stateT get_stateT put_stateT
    if monad return bind
```

[7] Such environments can be nicely handled by adding a reader monad transformer (§4).

The state monad transformer lifts the other effect operations fail, $\mathsf{¢}$, ask, tell, alt, altc, and pause according to their specifications. But catch cannot be lifted through stateT such that CATCH-GET and CATCH-PUT hold. As our exceptions carry no information, the inner monad cannot pass the state updates before the failure to the handler.

3.5 The Non-determinism Transformers

Non-determinism can be modelled by a collection type like lists, multisets, and sets. Thanks to value monomorphism, I can abstract over the collection type and provide one generic implementation. Later, I will obtain four implementations based on finite lists, finite multisets, finite sets, and countable sets by instantiation. Being finite, the first three only support binary non-deterministic choice alt, whereas countable sets additionally implement countable choice altc. Moreover, they all have an "empty" value—the empty list or (multi-)set—which can model failure. All implementations therefore provide a fail operation, which is the neutral element for nondeterministic choice: $\mathsf{alt}\ \mathsf{fail}\ m = m = \mathsf{alt}\ m\ \mathsf{fail}$. As I will discuss below, the different collection types impose different requirements on the inner monad.

The generic non-determinism transformer ndT changes the inner monad's value type from α to a collection of α, which I model by the type variable ζ. Thus, the inner monad's return operation has type $\zeta \Rightarrow \mu$ and bind has type $\mu \Rightarrow (\zeta \Rightarrow \mu) \Rightarrow \mu$. Similar to the other monad transformers, the $\mathsf{bind}_{\mathsf{ndT}}$ operation must first swap the inner monad constructor with the collection type constructor such that it can use the inner monad's bind operation. In my monomorphic setting, I model the swap as a merge operation with type $\zeta \Rightarrow (\alpha \Rightarrow \mu) \Rightarrow \mu$. It takes a collection C of values and a family f of non-deterministic computations indexed by C and merges all their effects and values into one computation. Moreover, I also need operations empty, single, and union (written infix as $\sqcup$) to construct collections, as I have abstracted over the concrete type. The locale $\mathsf{nondetM}$ captures these operations and their properties:

- The inner monad must be commutative (extension of the locale $\mathsf{monad\text{-}comm}$).
- The second argument to merge plays a role similar to the continuation arguments of other effect operations like get and $\mathsf{¢}$. Therefore, merge must respect the monad laws (MERGE-BIND and MERGE-RETURN).
- merge combines the effects of a computation family as expected for the collection operations (MERGE-EMPTY, MERGE-SINGLE, MERGE-UNION).
- The collection operations form a monoid, i.e., union is associative and empty the neutral element (MONOID).

$\mathtt{datatype}\ \mu\ \mathsf{ndT} = \mathsf{NdT}\ (\mathsf{run\text{-}nd} \colon \mu)$

$\mathtt{locale}\ \mathsf{nondetM} = \mathsf{monad\text{-}comm}\ \mathsf{return}\ \mathsf{bind}\ +$
$\quad \mathtt{fixes}\ \mathsf{merge} :: \zeta \Rightarrow (\alpha \Rightarrow \mu) \Rightarrow \mu$
$\quad\quad \mathsf{and}\ \mathsf{empty} :: \zeta\ \mathsf{and}\ \mathsf{single} :: \alpha \Rightarrow \zeta\ \mathsf{and}\ \mathsf{union} :: \zeta \Rightarrow \zeta \Rightarrow \zeta\ (\mathtt{infixl}\ \sqcup)$
$\quad \mathtt{assumes}\ \text{MERGE-BIND}:$
$\quad\quad\quad \mathsf{merge}\ C\ f \ggg\!= (\lambda C'.\ \mathsf{merge}\ C'\ g) = \mathsf{merge}\ C\ (\lambda x.\ f\ x \ggg\!= (\lambda C'.\ \mathsf{merge}\ C'\ g))$
$\quad\quad \mathsf{and}\ \text{MERGE-RETURN}:\ \mathsf{merge}\ C\ (\lambda x.\ \mathsf{return}\ (\mathsf{single}\ x)) = \mathsf{return}\ C$
$\quad\quad \mathsf{and}\ \text{MERGE-EMPTY}:\ \mathsf{merge}\ \mathsf{empty}\ f = \mathsf{return}\ \mathsf{empty}$
$\quad\quad \mathsf{and}\ \text{MERGE-SINGLE}:\ \mathsf{merge}\ (\mathsf{single}\ x)\ f = f\ x$
$\quad\quad \mathsf{and}\ \text{MERGE-UNION}:$
$\quad\quad\quad \mathsf{merge}\ (C \sqcup C')\ f = \mathsf{merge}\ C\ f \ggg\!= (\lambda A.\ \mathsf{merge}\ C'\ f \ggg\!= (\lambda A'.\ \mathsf{return}\ (A \sqcup A')))$
$\quad\quad \mathsf{and}\ \text{MONOID}:\ \mathsf{monoid}\ \mathsf{union}\ \mathsf{empty}$

I can now implement the monad operations for the non-determinism transformer for binary choice. Commutativity of the inner monad is needed only for ALT-BIND.

$\texttt{definition}\ \mathsf{return}_{\mathsf{ndT}} :: \alpha \Rightarrow \mu\ \mathsf{ndT}\ \texttt{where}$
$\quad \mathsf{return}_{\mathsf{ndT}}\ x = \mathsf{return}\ (\mathsf{single}\ x)$

$\texttt{definition}\ \mathsf{bind}_{\mathsf{ndT}} :: \mu\ \mathsf{ndT} \Rightarrow (\alpha \Rightarrow \mu\ \mathsf{ndT}) \Rightarrow \mu\ \mathsf{ndT}\ \texttt{where}$
$\quad m \gg\!=_{\mathsf{ndT}} f = \mathsf{NdT}\ (\mathsf{run\text{-}nd}\ m \gg\!= (\lambda C.\ \mathsf{merge}\ C\ (\mathsf{run\text{-}nd} \circ f)))$

$\texttt{definition}\ \mathsf{alt}_{\mathsf{ndT}} :: \mu\ \mathsf{ndT} \Rightarrow \mu\ \mathsf{ndT} \Rightarrow \mu\ \mathsf{ndT}\ \texttt{where}$
$\quad \mathsf{alt}_{\mathsf{ndT}}\ m_1\ m_2 = \mathsf{NdT}\ (\mathsf{run\text{-}nd}\ m_1 \gg\!= (\lambda C_1.\ \mathsf{run\text{-}nd}\ m_2 \gg\!= (\lambda C_2.\ \mathsf{return}\ (C_1 \sqcup C_2))))$

$\texttt{definition}\ \mathsf{fail}_{\mathsf{ndT}} :: \mu\ \mathsf{ndT}\ \texttt{where}$
$\quad \mathsf{fail}_{\mathsf{ndT}} = \mathsf{return}\ \mathsf{empty}$

$\texttt{lemma}\ \mathsf{monad\text{-}alt}\ \mathsf{return}_{\mathsf{ndT}}\ \mathsf{bind}_{\mathsf{ndT}}\ \mathsf{alt}_{\mathsf{ndT}}\ \texttt{and}\ \mathsf{monad\text{-}fail}\ \mathsf{return}_{\mathsf{ndT}}\ \mathsf{bind}_{\mathsf{ndT}}\ \mathsf{fail}_{\mathsf{ndT}}$

Most other effect operations lift through ndT as usual, except for catch and ¢. As $\mathsf{alt}_{\mathsf{ndT}}$ absorbs $\mathsf{fail}_{\mathsf{ndT}}$, failures cannot be caught. For ¢, BIND-SAMPLE$_2$ cannot be preserved: on the left-hand side, sampling from p is done independently for every possible result of m whereas on the right-hand side, the same sample y is used for all of m's results.

Countable choice $\mathsf{altc}_{\mathsf{ndT}}$ requires a bit more. Ideally, we could use merge to combine all the effects of countable choice, but the monomorphism restriction does not allow this: merge combines a family of computations indexed by a collection of *values*, whereas $\mathsf{altc}_{\mathsf{ndT}}$ must merge a family indexed by a countable set of *choices*. Like for probabilistic choices in §2.3, I do not want to unify the value type α with the choice type χ. I therefore fix another operation $\mathsf{merge}' :: \chi\ \mathsf{cset} \Rightarrow (\chi \Rightarrow \mu) \Rightarrow \mu$ and its appropriate properties. Then, I get

$\texttt{definition}\ \mathsf{altc}_{\mathsf{ndT}} :: \chi\ \mathsf{cset} \Rightarrow (\chi \Rightarrow \mu) \Rightarrow \mu\ \texttt{where}$
$\quad \mathsf{altc}_{\mathsf{ndT}}\ C\ f = \mathsf{NdT}\ (\mathsf{merge}'\ C\ (\mathsf{run\text{-}nd} \circ f))$

$\texttt{lemma}\ \mathsf{monad\text{-}altc}\ \mathsf{return}_{\mathsf{ndT}}\ \mathsf{bind}_{\mathsf{ndT}}\ \mathsf{altc}_{\mathsf{ndT}}$

Like for probabilistic sampling in §2.3, binary choice $\mathsf{alt}_{\mathsf{ndT}}$ can be expressed using countable choice $\mathsf{altc}_{\mathsf{ndT}}$ if the choice type χ contains at least three elements.

I have instantiated the generic implementation for four collection types: finite lists, finite multisets, finite sets, and countable sets. The first three are similar: The operations empty and single are obviously the empty and singleton list or (multi-)set, and union is list concatenation or (multi-)set union. Since the three collection types are all finite, merge is implemented by folding binary choice over the collection, starting with empty as the neutral element. For lists, e.g., I define $\mathsf{merge}_{\mathsf{list}}\ C\ f = \mathsf{foldr}\ (\lambda m_1\ m_2.\ m_1 \gg\!= (\lambda A.\ m_2 \gg\!= (\lambda B.\ \mathsf{return}\ (A +\!\!+ B))))\ (\mathsf{return}\ [])\ (\mathsf{map}\ f\ C)$, where foldr and map are the well-known functions on lists.

The requirements on the inner monad are as follows: Lists and multisets need a commutative monad, and finite sets need a commutative and duplicable monad. From a reasoning perspecitive, the list implementation is therefore inferior to multisets: they both require a commutative inner monad, but multisets satify more laws than lists. For example, $\mathsf{alt}_{\mathsf{multiset\ ndT}}$ is commutative, but $\mathsf{alt}_{\mathsf{list\ ndT}}$ is not. Conversely, from a programming perspective, the lack of commutativity allows a programmer to specify preferences among the alternatives, which cannot be done with (multi-)sets.

The finite set implementation is commutative if the inner monad is additionally discardable. Lists and multisets are not commutative for cardinality reasons. All implementations are neither discardable (because of $\mathsf{fail}_{\mathsf{ndT}}$) nor duplicable (because choices need not be made consistently.)

For countable sets, the inner monad must be commutative and duplicable. Yet, we cannot implement $\mathsf{merge}_{\mathsf{cset}}$ (or $\mathsf{merge}'_{\mathsf{cset}}$) using the operations of the inner monad as there is no "limit" operation to deal with infinite sets. Instead, I treat $\mathsf{merge}_{\mathsf{cset}}$ like another effect operation that monads can implement and transformers lift. Among the implementations, only the identity monad from §3.1 and the reader and failure transformers from §3.7 and §3.3 meet the commutativity and duplicatibility requirement. I have implemented the merge operations only for the identity monad and the reader transformer. Lifting fails for the failure transformer because the failure operation from the non-determinism transformer is incompatible with failure from the failure transformer.

3.6 Composing Monads with Transformers

Composing the two monad transformers failT and stateT with the monad prob, I can now instantiate the probabilistic interpreter from §2.4. As is well known, the order of composition matters. If I first apply failT to prob and then stateT (SFP for short), the resulting interpreter $\mathsf{eval}_{\mathrm{SFP}}\ E\ e :: (\nu \rightharpoonup \mathsf{int}, (\mathsf{int} \times (\nu \rightharpoonup \mathsf{int}))\ \mathsf{option}\ \mathsf{prob}\ \mathsf{failT})\ \mathsf{stateT}$ nests the result state of type $\nu \rightharpoonup \mathsf{int}$ inside the option type for failures, i.e., failures do not return a new state. Thus, failures erase state updates, i.e., $\mathsf{put}_{\mathrm{SFP}}\ s\ \mathsf{fail}_{\mathrm{SFP}} = \mathsf{fail}_{\mathrm{SFP}}$, and lazy and eager sampling are equivalent (LAZY-EAGER). Conversely, if I apply failT after stateT to prob (FSP for short), then $\mathsf{eval}_{\mathrm{FSP}}\ E\ e :: (\nu \rightharpoonup \mathsf{int}, (\mathsf{int}\ \mathsf{option} \times (\nu \rightharpoonup \mathsf{int}))\ \mathsf{prob})\ \mathsf{stateT}\ \mathsf{failT}$ and failures do return a new state as only the result type int sits inside option. In particular, $\mathsf{put}_{\mathrm{FSP}}\ s\ \mathsf{fail}_{\mathrm{FSP}} \neq \mathsf{fail}_{\mathrm{FSP}}$ in general, and lazy and eager sampling are not equivalent. I will consider the SFP case further in §4.

If we are interested in a non-deterministic rather than a probabilistic interpreter, then we can use the non-determinism monad transformer instead, say with countable choice. So let us compose countable choice cset ndT and stateT with the identity monad ident (SNI for short; the order NSI is not sensible as the non-determinism transformer should not be applied to a non-commutative state monad). Note that no failure transformer shows up in the composition as the non-determinism transformer already provides a failure operation. Failure therefore behaves differently from the probabilistic case. For example, I can define a lazy evaluator like in the probabilisitic case by using $\mathsf{choose\text{-}var}\ X\ x = \mathsf{altc}\ (X\ x)\ \mathsf{return}$ instead of sample-var. Consider the expression $e = \mathsf{Const}\ 2 \oslash \mathsf{Var}\ x_0$. Running $\mathsf{lazy}_{\mathrm{SNI}}\ X\ e$ in the initial state $\mathsf{empty} = (\lambda\_.\ \mathsf{None})$ with $X\ x = \{0,1\}$ yields only one possible outcome 2, which results from choosing 1 for x_0. Choosing 0 for x_0 results in a division by 0, i.e., a failure, and failures in ndT are silently ignored. In contrast, in the SFP monad, $\mathsf{lazy}_{\mathrm{SFP}}\ X'\ e$ with $X'\ x = \mathsf{uniform}\ \{0,1\}$ gives a uniform distribution over two outcomes: failure and 2. The SFP behaviour can be recovered by sandwiching a failure transformer between the state and non-determinism transformers (SFNI). Then, $\mathsf{lazy}_{\mathrm{SFNI}}\ X\ e$ also produces both outcomes, failure and 2.

3.7 Further Monads and Monad Transformers

Apart from the monad implementations presented so far, my library provides implementations for the other types of effects mentioned in §2.6. In particular, I define a reader (readT) and a writer monad transformer. The reader monad transformer differs from stateT only in that no updates are possible. Thus, $(\rho, \mu)\ \mathsf{readT}$ leaves the type of values of the inner monad unchanged, as no new state must be returned.

$\texttt{datatype}$ (ρ,μ) $\mathsf{readT} = \mathsf{ReadT}$ $(\mathsf{run\text{-}read}\colon \rho \Rightarrow \mu)$

$\texttt{context fixes}$ $\mathsf{return} :: \alpha \Rightarrow \mu$ $\mathsf{and\ bind} :: \mu \Rightarrow (\alpha \Rightarrow \mu) \Rightarrow \mu$

$\quad\texttt{definition}$ $\mathsf{return}_{\mathsf{readT}} :: \alpha \Rightarrow (\rho,\mu)\ \mathsf{readT}$ $\texttt{where}$

$\qquad\mathsf{return}_{\mathsf{readT}}\ x = \mathsf{ReadT}\ (\lambda\_.\ \mathsf{return}\ x)$

$\quad\texttt{definition}$ $\mathsf{bind}_{\mathsf{readT}} :: (\rho,\mu)\ \mathsf{readT} \Rightarrow (\alpha \Rightarrow (\rho,\mu)\ \mathsf{readT}) \Rightarrow (\rho,\mu)\ \mathsf{readT}$ $\texttt{where}$

$\qquad m \gg\!=_{\mathsf{readT}} f = \mathsf{ReadT}\ (\lambda r.\ \mathsf{run\text{-}read}\ m\ r \gg\!= (\lambda x.\ \mathsf{run\text{-}read}\ (f\ x)\ r))$

$\quad\texttt{definition}$ $\mathsf{ask}_{\mathsf{readT}} :: (\rho \Rightarrow (\rho,\mu)\ \mathsf{readT}) \Rightarrow (\rho,\mu)\ \mathsf{readT}$ $\texttt{where}$

$\qquad\mathsf{ask}_{\mathsf{readT}}\ f = \mathsf{ReadT}\ (\lambda r.\ \mathsf{run\text{-}read}\ (f\ r)\ r)$

$\quad\texttt{definition}$ $\mathsf{fail}_{\mathsf{readT}} :: (\mu \Rightarrow (\rho,\mu)\ \mathsf{readT})$ $\texttt{where}$ $\mathsf{fail}_{\mathsf{readT}}\ \mathsf{fail} = \mathsf{ReadT}\ (\lambda\_.\ \mathsf{fail})$

Resumptions are formalised as a plain monad using the codatatype

$\texttt{codatatype}$ (o,ι,α) $\mathsf{resumption} = \mathsf{Done}\ \alpha \mid \mathsf{Pause}\ o\ (\iota \Rightarrow (o,\iota,\alpha)\ \mathsf{resumption})$

Unfortunately, I cannot define resumptions as a monad transformer in HOL despite the restriction to monomorphic values. The reason is that for a transformer with inner monad τ, the second argument of the constructor Pause would have to be of type $\iota \Rightarrow (o,\iota,\alpha)\ \mathsf{resumption}\ \tau$, i.e., the codatatype would recurse through the unspecified type constructor τ. This is not supported by Isabelle's codatatype package [3] and, in fact, for some choices of τ, e.g., unbounded nondeterminism, the resumption transformer type does not exist in HOL at all. For the same reason, we cannot have other monad transformers that have similar recursive implementation types. Therefore, I fail to modularly construct all combinations of effects. For example, probabilistic resumptions with failures [22] are out of reach and must still be constructed from scratch.

3.8 Overloading the Monad Operations

When several monad transformers are composed, the monad operations quickly become large HOL terms as the transformer's operations take the inner monad's as explicit arguments. These large terms must be handled by the inference kernel, the type checker, the parser, and the pretty-printer, even if locale interpretations hide them from the user using abbreviations. To improve readability and the processing time of Isabelle, my library also defines the operations as single constants which are overloaded for the different monad implementations using recursion on types [35]. As overloading does not need these explicit arguments, it avoids the processing times for unification, type checking, and (un)folding of abbreviations. Yet, Isabelle's check against cyclic definitions [18] fails to see that the resulting dependencies must be acyclic (as the inner monad is always a type argument of the outer monad). So, I moved these overloaded definitions to a separate file and marked them as unchecked.[8] Overloading is just a syntactic convenience, on which the library and the examples in this paper do not rely. If users want to use it, they are responsible for not exploiting these unchecked dependencies.

---

[8] Isabelle's `adhoc-overloading` feature, which resolves overloading during type checking, cannot be used either as it does not support recursive resolutions. For example, resolving $\mathsf{return} :: \alpha \Rightarrow \alpha\ \mathsf{option\ ident\ failT}$ takes two steps: first to $\mathsf{return}_{\mathsf{failT}}\ \mathsf{return}$ and then to $\mathsf{return}_{\mathsf{failT}}\ \mathsf{return}_{\mathsf{ident}}$. The second step fails due to the intricate interleaving of type checking and resolution. Even if this is just an implementation issue, resolving overloading during type checking prevents definitions that are generic in the monad, which general overloading supports.

4 Moving Between Monad Instances

Once all variables have been sampled eagerly, the evaluation of the expression itself is deterministic. Thus, the actual evaluation need not be done in a monad as complex as FSP or SFP. It suffices to work in a reader-failure monad with operations fail and ask, which I obtain by applying the monad transformers readT and failT to ident (RFI for short). Such simpler monads have the advantage that reasoning becomes easier as more laws hold. I now explain how the theory of representation independence [25] can be used to move between different monad instances by going from SFP to RFI. This ultimately yields a theorem that characterises $\mathsf{eval}_{\mathrm{SFP}}$ in terms of $\mathsf{eval}_{\mathrm{RFI}}$. So, in general, this approach makes it possible to switch in the middle of a bigger proof from a complicated monad to a much simpler one.

Let me first deal with sampling. To go from α prob to β ident, I use a relation $\mathbb{IP}(A)$ between α ident and β prob since relations work better with higher-order functions than equations. Following Huffman and Kunčar [14], I call such a relation a correspondence relation. $\mathbb{IP}(A)$ is parametrised by a relation A between the values, which I will use later to express the differences in the values due to the monad transformers changing the value type of the inner monad. In detail, $\mathbb{IP}(A)$ relates a value $\mathsf{Ident}\ x$ to the one-point distribution $\mathsf{dirac}\ y$ iff A relates x to y. Then, the monad operations of ident and prob respect this relation. Respectfulness is formalised using the function relator $A \Mapsto B$, which was already used in §2.3 for parametricity. The following two conditions express that the monad operations respect $\mathbb{IP}(A)$:

- $(\mathsf{return}_{\mathsf{ident}}, \mathsf{return}_{\mathsf{prob}}) \in A \Mapsto \mathbb{IP}(A)$ and
- $(\mathsf{bind}_{\mathsf{ident}}, \mathsf{bind}_{\mathsf{prob}}) \in \mathbb{IP}(A) \Mapsto (A \Mapsto \mathbb{IP}(A)) \Mapsto \mathbb{IP}(A)$.

Note the similarity between the relations and the types of the monad operations, where A and $\mathbb{IP}$ take the roles of the type variables for values and of the monad type constructor, respectively. As the monad transformers failT and stateT are relationally parametric in the inner monad and eval is parametric in the monad, I prove the following relation between the evaluators automatically using Isabelle/HOL's Transfer prover [14]

$$(\mathsf{eval}_{\mathrm{SFP}}\ \mathsf{lookup}_{\mathrm{SFP}}\ e, \mathsf{eval}_{\mathrm{SFI}}\ \mathsf{lookup}_{\mathrm{SFI}}\ e) \in \mathsf{rel}_{\mathsf{stateT}}\ (\mathsf{rel}_{\mathsf{failT}}\ (\mathbb{IP}(=))) \qquad (4)$$

where SFI refers to the state-failure-identity composition of monads, and $\mathsf{rel}_{\mathsf{stateT}}$ and $\mathsf{rel}_{\mathsf{failT}}$ are the relators for the datatypes stateT and failT [3]. Formally, the relators lift relations on the inner monad to relations on the transformed monad. For example, $(m_1, m_2) \in \mathsf{rel}_{\mathsf{stateT}}\ M$ iff $(\mathsf{run\text{-}state}\ m_1\ s, \mathsf{run\text{-}state}\ m_2\ s) \in M$ for all s, and $(m_1, m_2) \in \mathsf{rel}_{\mathsf{failT}}\ M$ iff $(\mathsf{run\text{-}fail}\ m_1, \mathsf{run\text{-}fail}\ m_2) \in M$. Intuitively, (4) states that in the monads SFP and SFI, eval behaves the same with respect to states updates and failure and the results are the same; in particular, the evaluation is deterministic.

In the following, I use the property of a relator rel that if M is the graph $\mathsf{Gr}\ f$ of a function f, then $\mathsf{rel}\ M$ is the graph of the function $\mathsf{map}\ f$, where map is the canonical map function for the relator. For example, $\mathsf{map}_{\mathsf{failT}}\ f = \mathsf{FailT} \circ f \circ \mathsf{run\text{-}fail}$, so

$$\mathsf{rel}_{\mathsf{failT}}\ (\mathsf{Gr}\ f) = \mathsf{Gr}\ (\mathsf{map}_{\mathsf{failT}}\ f) \qquad (5)$$

where $(x, y) \in \mathsf{Gr}\ f$ iff $f\ x = y$. Isabelle's datatype package automatically proves these relator-graph identities. The correspondence relation $\mathbb{IP}$ satisfies a similar law: $\mathbb{IP}(\mathsf{Gr}\ f) = \mathsf{Gr}\ (\mathsf{map}_{\mathbb{IP}}\ f)$ where $\mathsf{map}_{\mathbb{IP}}\ f\ = \mathsf{dirac} \circ f \circ \mathsf{run\text{-}ident}$.

Having eliminated probabilities, I next switch from the state monad transformer to the reader monad transformer. I again define a correspondence relation $\mathbb{RS}(s, M)$ between readT and stateT. It takes as parameters the environment s and the correspondence relation M between the inner monads. It relates the two monadic values m_1 and m_2 iff M relates the results of running m_1 and m_2 on s, i.e., $(\mathsf{run\text{-}read}\ m_1\ s, \mathsf{run\text{-}state}\ m_2\ s) \in M$. Again, I show that the monad operations respect $\mathbb{RS}(s, M)$ as formalised below. As readT and stateT are monad transformers, I assume that the operations of the inner monads respect M. These assumptions can be expressed using $\Rrightarrow$ since the inner operations are arguments to readT's and stateT's operations. Here, $A \lessdot\!\!\ltimes s$ adapts the relation A on values to stateT's change of the value type from α to $\alpha \times \sigma$; $(x, (y, s')) \in A \lessdot\!\!\ltimes s$ iff $(x, y) \in A$ and $s' = s$, i.e., A relates the results and the state is not updated.

- $(\mathsf{return}_{\mathsf{readT}}, \mathsf{return}_{\mathsf{stateT}}) \in (A \lessdot\!\!\ltimes s \Rrightarrow M) \Rrightarrow A \Rrightarrow \mathbb{RS}(s, M)$,
- $(\mathsf{bind}_{\mathsf{readT}}, \mathsf{bind}_{\mathsf{stateT}}) \in$
  $(M \Rrightarrow (A \lessdot\!\!\ltimes s \Rrightarrow M) \Rrightarrow M) \Rrightarrow \mathbb{RS}(s, M) \Rrightarrow (A \Rrightarrow \mathbb{RS}(s, M)) \Rrightarrow \mathbb{RS}(s, M)$,
- $(\mathsf{ask}_{\mathsf{readT}}, \mathsf{get}_{\mathsf{stateT}}) \in (\{(s, s)\} \Rrightarrow \mathbb{RS}(s, M)) \Rrightarrow \mathbb{RS}(s, M)$, and
- $(\mathsf{fail}_{\mathsf{readT}}, \mathsf{fail}_{\mathsf{stateT}}) \in M \Rrightarrow \mathbb{RS}(s, M)$,

Then, by representation independence, the Transfer package automatically proves the following relation between $\mathsf{eval}_{\mathrm{RFI}}$ and $\mathsf{eval}_{\mathrm{SFI}}$, where $\mathsf{lookup}_{\mathrm{RFI}}$ uses $\mathsf{ask}_{\mathsf{readT}}$ instead of $\mathsf{get}_{\mathsf{stateT}}$, and $\mathsf{rel}_{\mathsf{ident}}$ and $\mathsf{rel}_{\mathsf{option}}$ are the relators for the datatypes ident and option.

$$(\mathsf{eval}_{\mathrm{RFI}}\ \mathsf{lookup}_{\mathrm{RFI}}\ e, \mathsf{eval}_{\mathrm{SFI}}\ \mathsf{lookup}_{\mathrm{SFI}}\ e) \in \mathbb{RS}(s, \mathsf{rel}_{\mathsf{failT}}\ (\mathsf{rel}_{\mathsf{ident}}\ (\mathsf{rel}_{\mathsf{option}}\ (= \lessdot\!\!\ltimes s))))$$

This says that running eval in RFI and SFI computes the same result, has the same behaviour with respect to state queries and failures, and does not update the state.

Actually, I can go from SFP directly to RFI, without the monad SFI as a stepping stone, thanks to $\mathbb{IP}$ taking a relation on the value types:

$$(\mathsf{eval}_{\mathrm{RFI}}\ \mathsf{lookup}_{\mathrm{RFI}}\ e, \mathsf{eval}_{\mathrm{SFP}}\ \mathsf{lookup}_{\mathrm{SFP}}\ e) \in \mathbb{RS}(s, \mathsf{rel}_{\mathsf{failT}}\ (\mathbb{IP}(\mathsf{rel}_{\mathsf{option}}\ (= \lessdot\!\!\ltimes s)))) \quad (6)$$

As $= \lessdot\!\!\ltimes s$ is the graph of $\lambda a.\ (a, s)$, using only the graph properties like (5) of $\mathbb{IP}$ and the relators, and using $\mathbb{RS}$'s definition, I derive the characterisation of $\mathsf{eval}_{\mathrm{SFP}}$ from (6):

$$\begin{aligned}&\mathsf{run\text{-}state}\ (\mathsf{eval}_{\mathrm{SFP}}\ \mathsf{lookup}_{\mathrm{SFP}}\ e)\ s = \\ &\qquad \mathsf{map}_{\mathsf{failT}}\ (\mathsf{map}_{\mathbb{IP}}\ (\mathsf{map}_{\mathsf{option}}\ (\lambda a.\ (a, s))))\ (\mathsf{run\text{-}read}\ (\mathsf{eval}_{\mathrm{RFI}}\ \mathsf{lookup}_{\mathrm{RFI}}\ e)\ s)\end{aligned}$$

where $\mathsf{map}_{\mathsf{failT}}$ and $\mathsf{map}_{\mathsf{option}}$ are the canonical map functions for failT and option. Thus, instead of reasoning about $\mathsf{eval}_{\mathrm{SFP}}$ in SFP, I can conduct the proofs in the simpler monad RFI. For example, as RFI is commutative, subexpressions can be evaluated in any order. Thus, I get the following identity expressing the reversed evaluation order (and a similar one for $\oslash$).[9]

$$\mathsf{eval}_{\mathrm{RFI}}\, E\, (e_1 \oplus e_2) = \mathsf{eval}_{\mathrm{RFI}}\, E\, e_2 \mathbin{\gg\!=}_{\mathrm{RFI}} (\lambda j.\, \mathsf{eval}_{\mathrm{RFI}}\, E\, e_1 \mathbin{\gg\!=}_{\mathrm{RFI}} (\lambda i.\, \mathsf{return}_{\mathrm{RFI}}\, (i + j)))$$

In summary, I have demonstrated a generic approach to switch from a complicated monad to a much simpler one. Conceptually, the correspondence relations $\mathbb{IP}$ and $\mathbb{RS}$ just embed one monad or monad transformer (ident and readT) in a richer one (prob and stateT). It is precisely this embedding that ultimately yields the map functions in the characterisation. In this functional view, the respectfulness conditions express that the embedding is a monad homomorphism. Yet, I use relations for the embedding instead of functions because only relations work for higher-order operations in a compositional way.

[9] Following the "as abstract as possible" spirit of this paper, I actually proved the identities in the locale of commutative monads and showed that readT is commutative if its inner monad is.

The reader may wonder why I go through all the trouble of defining correspondence relations and showing respectfulness and parametricity. Indeed, in this example, it would probably have been easier to simply perform an induction over expressions and prove the equation directly. The advantage of my approach is that it does not rely on the concrete definition of eval. It suffices to know that eval is parametric in the monad, which Isabelle derives automatically from the definition. This automated approach therefore scales to arbitrarily complicated monadic functions whereas induction proofs do not. Moreover, note that the correspondence relations and respectfulness lemmas only depend on the monads. They can therefore be reused for other monadic functions.

5 Related work

Huffman et al. [13, 15] formalise the concept of value-polymorphic monads and several monad transformers in Isabelle/HOLCF, the domain theory library of Isabelle/HOL. They circumvent HOL's type system restrictions by projecting everything into HOLCF's universal domain of computable values. That is, they trade in HOL's set-theoretic model with its simple reasoning rules for a domain-theoretic model with ubiquituous $\perp$ values and strictness side conditions. This way, they can define a resumption monad transformer (for computable continuations). Being tied to domain theory, their library cannot be used to model effects of *plain* HOL functions, which is my goal, the strictness assumptions make their laws and proofs more complicated than mine, and functions defined with HOLCF do not work with Isabelle's code generator. Still, their idea of projecting everything into a universal type could also be adapted to plain HOL, albeit only for a restricted class of monads; achieving a similar level of automation and modularity would require a lot more effort than my approach, which uses only existing Isabelle features.

Gibbons and Hinze [7] axiomatize monads and effects using Haskell-style type constructor classes and use the algebraic specification to prove identities between Haskell programs, similar to my abstract locales in §2. Their specification of state effects omits GET-CONST, but they later assume that it holds [7, §10.2]. Being value-polymorphic, their operations do not need my continuations and the laws are therefore simpler. In particular, no new assumptions are typically needed when monad specifications are combined. In contrast, my continuations sometimes require interaction assumptions like SAMPLE-GET. Gibbons and Hinze only consider reasoning in the abstract setting and do not discuss the transition to concrete implementations and the relations between implementations. Also, they do not prove that monad implementations satisfy their specifications. Later, Jeuring et al. [17] showed that the implementations in Haskell do not satisfy them because of strictness issues similar to the ones in Huffman's work.

Lobo Vesga [21] formalised some of Gibbons' and Hinze's examples in Agda. She does not need assumptions for the continuations like I do as value-polymorphic monads can be directly expressed in Agda. Like Gibbons and Hinze, she does not study the connection between specifications and implementations. Thanks to the good proof automation in Isabelle, my mechanised proofs are much shorter than hers, which are as detailed as Gibbons' and Hinze's pen-and-paper proofs.

Lochbihler and Schneider [24] implemented support for equational reasoning about applicative functors, which are more general than monads. They focus on lifting identities on values to a concrete applicative functor. Reasoning with abstract applicative functors is not supported. Like monads, the concept of an applicative functor cannot be expressed as a predicate in HOL. Moreover, the applicative operations do not admit value monomorphisation like monads do, as the type of $\diamond$ contains applications of the functor type constructor τ to $\alpha \Rightarrow \beta$, α, and β.[10] So, monads seem to be the right choice, even though I could have defined the interpreter eval applicatively (but not, e.g., memoisation).

Grimm et al. [8] model several effects and their combinations in the dependently typed logic of F* to reason about various effectful programs. They need not choose between effect and value polymorphism thanks to F*'s richer logic. Like me, they model monads for probabilities, state, exceptions, and the reader monad, and study among others the memoization problem and an interpreter. They also discuss how they can switch from a reader monad to a state monad. Yet, their definitions do not achieve my level of modularity in two respects: First, the type annotation of an F* function fixes the monad implementation whereas my approach with locales can leave the implementation abstract. Second, they define a new monad for every effect combination whereas I combine effects modularly thanks to monad transformers.

Wimmer et al. [36] propose a tool to automatically memoize pure recursive functions using a state monad similar to my memo function. They use similar ideas of relational parametricity (§4) to prove the monadification step correct. However, their memoization function only works in a concrete state monad and only for pure functions; other effects like probabilistic choice and non-determinism are not yet supported.

6 Conclusion

I have presented a library of abstract monadic effect specifications and their implementations as monads and monad transformers in Isabelle/HOL. I illustrated its usage and the elegance of reasoning using a monadic interpreter. The type system of HOL forced me to restrict the monads to monomorphic values. Monomorphic values work well when the reasoning involves only a few monadic functions like in the running example. In larger projects, this restriction can become a limiting factor. Nevertheless, in a project on formalising computational soundness results,[11] I successfully formalised and reasoned about several complicated serialisers and parsers for symbolic messages of security protocols. In that work, reasoning abstractly about effects and being able to move from one monad instance to another were crucial. For example, the serialiser converts symbolic protocol messages into bitstrings. The challenges were similar to those of the interpreter eval. Serialisation may fail when the symbolic message is not well-formed, similar to division by zero in the interpreter. When serialisation encounters a new nonce, it randomly samples a fresh bitstring, which must also be used for serialising further occurrences of the same nonce. I formalised this similar to the memoisation of variable evaluation in the interpreter. A further challenge not present in the interpreter was that the serialiser must also

[10] The alternative applicative interface [16] with the operator $\mathsf{zip} :: \alpha\ \tau \Rightarrow \beta\ \tau \Rightarrow (\alpha \times \beta)\ \tau$ is amenable to monomorphisation if we restrict ourselves to *infinite* value types α as then $\alpha \times \alpha$ is isomorphic to a. This interface is tailored towards a first-order language [10]. So functions must be uncurried and their arguments encoded using the isomorphism. We would thus clutter the definitions and proofs with conversions and lose the benefits of built-in higher-order unification.

[11] http://www.infsec.ethz.ch/research/projects/FCSPI.html

record the serialisation of all subexpressions such that the parser can map bitstrings generated by the serialiser back to symbolic messages without calling a decryption oracle or inverting a cryptographic hash function. The construction relied on the invariant that the recorded values were indeed generated by the serialiser, but such an invariant cannot be expressed easily for a probabilistic, stateful function. I therefore formalised also the switch from lazy to eager sampling for the serialiser (lazy sampling was needed to push the randomisation of encryptions into an encryption oracle) and the switch to a read-only version without recording of results using techniques similar to the example in §4.

Instead of specifying effects abstractly and composing them using monad transformers, I obviously could have formalised everything in a sufficiently rich monad that covers all the effects of interest, e.g., continuations. Then, there would be no need for abstract specifications as I could work directly with a concrete monad as usual, where my reasoning on the abstract level could be mimicked. But I would deprive myself of the option of going to a specific monad that covers precisely the effects needed. Such specialisation has two advantages: First, as shown in §4, simpler monads satisfy more laws, e.g., commutativity, which make the proofs easier. Second, concrete monads can have dedicated setups for reasoning and proof automation that are not available in the abstract setting. My library achieves the best of both worlds. I can reason abstractly and thus achieve generality. When this gets too cumbersome or impossible, I can switch to a concrete monad, continuing to use the abstract properties already proven.

In the long run, I can imagine a definitional package for monads and monad transformers that composes concrete value-polymorphic monad transformers. Similar to how Isabelle's datatype package composes bounded natural functors [3], such a package must perform the construction and the derivation of all laws afresh for every concrete combination of monads, as value-polymorphic monads lie beyond HOL's expressiveness. When combined with a reinterpretation framework for theories, I could model effects and reason about them abstractly and concretely without the restriction to monomorphic values.

Acknowledgements I thank Dmitriy Traytel and the anonymous reviewers for suggesting many improvements to the presentation. This work is supported by the Swiss National Science Foundation grant 153217 "Formalising Computational Soundness for Protocol Implementations".

References

1. Back, R.J., Wright, J.: Refinement Calculus–A Systematic Introduction. Springer (1998)
2. Ballarin, C.: Locales: A module system for mathematical theories. J. Automat. Reason. **52**(2), 123–153 (2014). DOI 10.1007/s10817-013-9284-7
3. Blanchette, J.C., Hölzl, J., Lochbihler, A., Panny, L., Popescu, A., Traytel, D.: Truly modular (co)datatypes for Isabelle/HOL. In: ITP 2014, *LNCS*, vol. 8558, pp. 93–110. Springer (2014)
4. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: TPHOLs 2008, *LNCS*, vol. 5170, pp. 134–149. Springer (2008). DOI 10.1007/978-3-540-71067-7_14
5. Eberl, M., Hölzl, J., Nipkow, T.: A verified compiler for probability density functions. In: J. Vitek (ed.) ESOP 2015, *LNCS*, vol. 9032, pp. 80–104. Springer (2015). DOI 10.1007/978-3-662-46669-8_4

6. Erwig, M., Kollmansberger, S.: Functional pearls: Probabilistic functional programming in Haskell. Journal of Functional Programming **16**, 21–34 (2006). DOI 10.1017/S0956796805005721
7. Gibbons, J., Hinze, R.: Just do it: Simple monadic equational reasoning. In: ICFP 2011, pp. 2–14. ACM (2011). DOI 10.1145/2034773.2034777
8. Grimm, N., Maillard, K., Fournet, C., Hriţcu, C., Maffei, M., Protzenko, J., Ramananandro, T., Rastogi, A., Swamy, N., Zanella Béguelin, S.: A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations. In: CPP 2018, pp. 130–145. ACM (2018). DOI 10.1145/3167090
9. Harrison, W.L.: The essence of multitasking. In: M. Johnson, V. Vene (eds.) Algebraic Methodology and Software Technology (AMAST 2006), *LNCS*, vol. 4019, pp. 158–172. Springer (2006). DOI 10.1007/11784180_14
10. Hinze, R.: Lifting operators and laws. http://www.cs.ox.ac.uk/ralf.hinze/Lifting.pdf (2010)
11. Hölzl, J., Lochbihler, A., Traytel, D.: A formalized hierarchy of probabilistic system types. In: ITP 2015, *LNCS*, vol. 9236, pp. 203–220. Springer (2015). DOI 10.1007/978-3-319-22102-1_13
12. Homeier, P.V.: The HOL-Omega logic. In: S. Berghofer, T. Nipkow, C. Urban, M. Wenzel (eds.) TPHOLs 2009, *LNCS*, vol. 5674, pp. 244–259. Springer (2009). DOI 10.1007/978-3-642-03359-9_18
13. Huffman, B.: Formal verification of monad transformers. In: ICFP 2012, pp. 15–16. ACM (2012). DOI 10.1145/2364527.2364532
14. Huffman, B., Kunčar, O.: Lifting and Transfer: A modular design for quotients in Isabelle/HOL. In: CPP 2013, *LNCS*, vol. 8307, pp. 131–146. Springer (2013). DOI 10.1007/978-3-319-03545-1_9
15. Huffman, B., Matthews, J., White, P.: Axiomatic constructor classes in Isabelle/HOLCF. In: J. Hurd, T. Melham (eds.) TPHOLs 2005, *LNCS*, vol. 3603, pp. 147–162. Springer (2005). DOI 10.1007/11541868_10
16. Hutton, G.: Higher-order functions for parsing. Journal of Functional Programming **2**(3), 323–343 (1992)
17. Jeuring, J., Jansson, P., Amaral, C.: Testing type class laws. In: Haskell 2012, pp. 49–60. ACM (2012). DOI 10.1145/2364506.2364514
18. Kunčar, O.: Correctness of Isabelle's cyclicity checker: Implementability of overloading in proof assistants. In: CPP 2015, pp. 85–94. ACM (2015). DOI 10.1145/2676724.2693175
19. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: ITP 2012, *LNCS*, vol. 7406, pp. 166–182. Springer (2012). DOI 10.1007/978-3-642-32347-8_12
20. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: POPL 1995, pp. 333–343. ACM (1995). DOI 10.1145/199448.199528
21. Lobo Vesga, E.: Hacia la formalización del razonamiento ecuacional sobre mónadas. Tech. rep., Universidad EAFIT (2013). http://hdl.handle.net/10784/4554
22. Lochbihler, A.: Probabilistic functions and cryptographic oracles in higher order logic. In: P. Thiemann (ed.) ESOP 2016, *LNCS*, vol. 9632, pp. 503–531. Springer (2016). DOI 10.1007/978-3-662-49498-1_20
23. Lochbihler, A.: Effect polymorphism in higher-order logic (proof pearl). In: M. Ayala-Rincón, C.A. Muñoz (eds.) Interactive Theorem Proving (ITP 2017), *LNCS*, vol. 10499, pp. 389–409. Springer (2017). DOI 10.1007/978-3-319-66107-0_25
24. Lochbihler, A., Schneider, J.: Equational reasoning with applicative functors. In: J.C. Blanchette, S. Merz (eds.) ITP 2016, *LNCS*, vol. 9807, pp. 252–273. Springer (2016). DOI 10.1007/978-3-319-43144-4_16
25. Mitchell, J.C.: Representation independence and data abstraction. In: POPL 1986, pp. 263–276. ACM (1986). DOI 10.1145/512644.512669
26. Moggi, E.: An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, LFCS, School of Informatics, University of Edinburgh (1990)
27. Nipkow, T., Klein, G.: Concrete Semantics. Springer (2014). DOI 10.1007/978-3-319-10542-0
28. Nipkow, T., Paulson, L.C.: Proof pearl: Defining functions over finite sets. In: J. Hurd, T. Melham (eds.) TPHOLs 2005, *LNCS*, vol. 3603, pp. 385–396. Springer (2005)
29. Piróg, M., Gibbons, J.: The coinductive resumption monad. In: Mathematical Foundations of Programming Semantics (MFPS 2014), *ENTCS*, vol. 308, pp. 273–288 (2014). DOI 10.1016/j.entcs.2014.10.015

30. Ramsey, N., Pfeffer, A.: Stochastic lambda calculus and monads of probability distributions. In: POPL 2002, pp. 154–165. ACM (2002). DOI 10.1145/503272.503288
31. Reynolds, J.C.: Types, abstraction and parametric polymorphism. In: IFIP 1983, *Information Processing*, vol. 83, pp. 513–523. North-Holland/IFIP (1983)
32. Sternagel, C., Thiemann, R.: A framework for developing stand-alone certifiers. In: M. Ayala-Rincón, I. Mackie (eds.) Logical and Semantic Frameworks with Applications (LSFA 2014), *ENTCS*, vol. 312, pp. 51–67 (2015). DOI 10.1016/j.entcs.2015.04.004
33. Wadler, P.: How to replace failure by a list of successes: a method for exception handling, backtracking, and pattern matching in lazy functional languages. In: J.P. Jouannaud (ed.) Functional Programming Languages and Computer Architecture (FPCA 1985), *LNCS*, vol. 201, pp. 113–128. Springer (1985). DOI https://doi.org/10.1007/3-540-15975-4_33
34. Wadler, P.: Monads for functional programming. In: J. Jeuring, E. Meijer (eds.) Advanced Functional Programming, *LNCS*, vol. 925, pp. 24–52. Springer (1995). DOI 10.1007/3-540-59451-5_2
35. Wenzel, M.: Type classes and overloading in higher-order logic. In: E.L. Gunter, A. Felty (eds.) TPHOLs 1997, *LNCS*, vol. 1275, pp. 307–322. Springer (1997). DOI 10.1007/BFb0028402
36. Wimmer, S., Hu, S., Nipkow, T.: Verified memoization and dynamic programming. In: ITP 2018, LNCS. Springer (2018). To appear.

A Step-By-Step Proof of Lemma MEMO-IDEM

Proof First, I prove that updating the table of memoised calls is idempotent. Let $\mathsf{U}\ x\ y = \mathsf{update}\ (\lambda t.\ t(x \mapsto y))\ (\mathsf{return}\ y)$. Then, $\mathsf{U}\ x\ y \gg\!= \mathsf{U}\ x = \mathsf{U}\ x\ y$ holds:

$$\begin{aligned}
& \mathsf{U}\ x\ y \gg\!= \mathsf{U}\ x = \mathsf{update}\ (\lambda t.\ t(x \mapsto y))\ (\mathsf{return}\ y) \gg\!= U\ x \\
= \{ & \text{ UPDATE-BIND } \} \\
& \mathsf{update}\ (\lambda t.\ t(x \mapsto y))\ (\mathsf{return}\ y \gg\!= \mathsf{U}\ x) \\
= \{ & \text{ RETURN-BIND, definition of } \mathsf{U} \} \\
& \mathsf{update}\ (\lambda t.\ t(x \mapsto y))\ (\mathsf{update}\ (\lambda t.\ t(x \mapsto y))\ (\mathsf{return}\ y)) \\
= \{ & \text{ UPDATE-UPDATE } \} \\
& \mathsf{update}\ ((\lambda t.\ t(x \mapsto y)) \circ (\lambda t.\ t(x \mapsto y)))\ (\mathsf{return}\ y) \\
= \{ & \text{ idempotence of } \lambda t.\ t(x \mapsto y) \} \\
& \mathsf{update}\ (\lambda t.\ t(x \mapsto y))\ (\mathsf{return}\ y) = \mathsf{U}\ x\ y
\end{aligned}$$

Next, let $\mathsf{F} = \lambda \mathit{table}'.\ \mathsf{case}\ \mathit{table}'\ x\ \mathsf{of\ Some}\ y \Rightarrow \mathsf{return}\ y \mid \mathsf{None} \Rightarrow f\ x \gg\!= \mathsf{U}\ x$. Then,

$$\begin{aligned}
& \mathsf{memo}\ (\mathsf{memo}\ f)\ x \\
= \{ & \text{ definition of } \mathsf{memo} \} \\
& \mathsf{get}\ (\lambda \mathit{table}.\ \mathsf{case}\ \mathit{table}\ x\ \mathsf{of\ Some}\ y \Rightarrow \mathsf{return}\ y \\
& \qquad\qquad\qquad\qquad\qquad \mid \mathsf{None} \Rightarrow \mathsf{get}\ \mathsf{F} \gg\!= \mathsf{U}\ x) \\
= \{ & \text{ BIND-GET, GET-CONST } \} \\
& \mathsf{get}\ (\lambda \mathit{table}.\ \mathsf{case}\ \mathit{table}\ x\ \mathsf{of\ Some}\ y \Rightarrow \mathsf{get}\ (\lambda_.\ \mathsf{return}\ y) \\
& \qquad\qquad\qquad\qquad\qquad \mid \mathsf{None} \Rightarrow \mathsf{get}\ (\lambda \mathit{table}'.\ \mathsf{F}\ \mathit{table}' \gg\!= \mathsf{U}\ x)) \\
= \{ & \text{ } \mathsf{case} \text{ distributes over } \mathsf{get} \} \\
& \mathsf{get}\ (\lambda \mathit{table}.\ \mathsf{get}\ (\lambda \mathit{table}'.\ \mathsf{case}\ \mathit{table}\ x\ \mathsf{of\ Some}\ y \Rightarrow \mathsf{return}\ y \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mid \mathsf{None} \Rightarrow \mathsf{F}\ \mathit{table}' \gg\!= \mathsf{U}\ x)) \\
= \{ & \text{ GET-GET } \} \\
& \mathsf{get}\ (\lambda \mathit{table}.\ \mathsf{case}\ \mathit{table}\ x\ \mathsf{of\ Some}\ y \Rightarrow \mathsf{return}\ y \mid \mathsf{None} \Rightarrow \mathsf{F}\ \mathit{table} \gg\!= \mathsf{U}\ x) \\
= \{ & \text{ propagate } \mathit{table}\ x = \mathsf{None} \text{ into } \mathsf{F} \} \\
& \mathsf{get}\ (\lambda \mathit{table}.\ \mathsf{case}\ \mathit{table}\ x\ \mathsf{of\ Some}\ y \Rightarrow \mathsf{return}\ y \mid \mathsf{None} \Rightarrow (f\ x \gg\!= \mathsf{U}\ x) \gg\!= \mathsf{U}\ x) \\
= \{ & \text{ BIND-ASSOC, } \mathsf{U} \text{ idempotent, definition of } \mathsf{memo} \} \\
& \mathsf{memo}\ f\ x
\end{aligned}$$

□