Abstract

Proof assistants based on dependent type theory are closely related to functional programming languages, and so it is tempting to use them to prove the correctness of functional programs. In this paper, we show how Agda, such a proof assistant, can be used to prove theorems about Haskell programs. Haskell programs are translated into an Agda model of their semantics, by translating via GHC’s Core language into a monadic form specially adapted to represent Haskell’s polymorphism in Agda’s predicative type system. The translation can support reasoning about either total values only, or total and partial values, by instantiating the monad appropriately. We claim that, although these Agda models are generated by a relatively complex translation process, proofs about them are simple and natural, and we offer a number of examples to support this claim.

Categories and Subject Descriptors D.1.1 [Programming Techniques]: Applicative (Functional) Programming; D.2.4 [Software Engineering]: Software/Program Verification—Correctness proofs; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—Mechanical verification

General Terms Languages, Theory, Verification

Keywords Haskell, GHC Core, Type Theory, Monadic Translation, Partiality, Verification

1. Introduction

Constructive type theories (see for example [16, 6]) have long been touted as a promising approach to writing correct software. These are type systems with dependent types, in which propositions can be represented as types via the Curry-Howard isomorphism [11], and constructive proofs of those propositions can be represented as terms of the corresponding types. Several proof editors (Agda [5], Coq [2], Twelf [17]) have been developed based on such theories; they interact with users to construct a term (proof) of a given goal type, ensuring that type correctness is preserved at each step, and so the proof constructed is valid. In this paper, we show how Agda can be used to develop verified Haskell programs.

The traditional approach to developing verified programs using type theory, is to extract them from proofs. One begins by expressing a specification as a type; for example,

\[
\forall x : \text{List Integer}. \forall y : \text{List Integer}. \quad \text{isPermutation}(x, y) \land \text{isOrdered}(y)
\]

says that sorting is possible. A term of this type contains an embedded sorting algorithm, together with proof fragments. Program extraction discards these fragments, generating a verified sorting function as its result. Program extraction has been implemented in the Coq system, generating programs in OCaml, Caml Light, or Haskell, and used to construct verified programs of many hundred lines.

However, this approach does demand an all-or-nothing commitment to a new programming method. One begins by formalising a specification, devotes much subsequent work to proof, and only in the final stages obtains a program which can actually be run. What if the specification proves to be wrong, and the error is only revealed when the generated program doesn’t behave as the user (informally) expected? Then much work has been wasted, and this work is difficult to reuse. While specifications for small functions like sorting are easy to get right, in more realistic situations they are likely to be wrong. Our own experience using our random testing tool QuickCheck [4], which tests Haskell programs against specifications to reveal errors in both, is that errors in specifications are just as common as errors in programs. In industrial projects, specifications change constantly. We believe, therefore, that the program extraction approach will be difficult to scale up to realistic applications.

The alternative approach we propose is to develop programs by combining proof with testing. We start by writing programs and testing them as usual. Then we develop specifications in the form of properties which are tested against the program by QuickCheck. At this stage, most inconsistencies between the code and its specification are revealed cheaply. Only once testing reveals no further errors, do we go on to prove the most important properties using Agda. At this stage, the proofs are likely to succeed—which is important, because attempting a proof is in general a very costly way to find a mistake. With this approach, we spend the effort of formal proof only where it is most needed, which should make the method as a whole more suitable for deployment in practice.

Although our approach may seem less than “purist”, we may liken this way of working to that of a mathematician who studies examples, hypotheses, and counter-examples, before embarking on the hard work of formulating theorems and finding proofs—which is, of course, the way mathematicians work in reality!

However, the critical point here is that, unlike with the program extraction approach, the Haskell code to be verified exists before we start proving. Thus we must import existing Haskell code into
the prover, unlike program extraction, which need only export code from the prover (a process which is not provided by all proof assistants, and whose correctness is usually not verified!). Agda is designed to use a syntax similar to Haskell, but we cannot simply take the Haskell program and supply it as input to Agda because the semantics differs in important ways. Hitherto Agda users have translated programs to be verified into the Agda language by hand, but on a larger scale such hand modelling is not reliable; translating thousands of lines of code by hand would certainly introduce errors, defeating the whole purpose of formal verification. Thus, to make our approach work, we must develop a translator which automatically converts Haskell programs into a suitable Agda model.

Such a translation is more difficult than it seems. The major constraint is that the user of the theorem prover must be able to prove properties of the translated code. These proofs must be reasonably elegant, not cluttered with detail introduced by the translation. Moreover, since reading machine generated code is, in general, an unpleasant experience, we aim to make it possible to prove properties of the translated code without reading it—it should be sufficient to refer to the Haskell source itself, to understand how proofs should be constructed. These constraints strongly influence our choice of translation. Of course, we want to exploit the deep similarities between Haskell and Agda, so that the translation resembles a “natural” Agda model, but there are fundamental differences to be overcome, caused by the differing requirements on a programming language and a proof language.

In this paper, we present the translation method we have developed, together with applications to small programs to justify our claim that proofs about translated code are quite natural. While many problems remain to be solved, we do support a large subset of Haskell, and we address the fundamental problem of partiality—Haskell programs may loop or fail, while Agda programs, by definition, must not.

Figure 1 gives an overview over our translation: A Haskell program is first translated into Haskell Core language via the Glasgow Haskell Compiler (GHC). Then a preprocessor classifies types into monomorphic and polymorphic types. From that, the monadic translation produces Agda code parametrized by a monad, which can be instantiated to the identity monad if one wants to prove program properties under the assumption that all objects are total, or to the Maybe monad if one takes also partial objects into consideration.

As a simple example, we shall prove properties of the queue implementation in Figure 2. This implements queues efficiently as pairs of lists, the “front” and the “back”, with the back held in reverse order. The stated properties relate this efficient implementation to an abstract model where a queue is just a list of elements. The properties have been tested by QuickCheck: we will show in Section 5.2 how they can also be proved using Agda.

As a simple example, we shall prove properties of the queue implementation in Figure 2. This implements queues efficiently as pairs of lists, the “front” and the “back”, with the back held in reverse order. The stated properties relate this efficient implementation to an abstract model where a queue is just a list of elements. The properties have been tested by QuickCheck: we will show in Section 5.2 how they can also be proved using Agda.

The rest of the paper is structured as follows. In Section 2 we give an overview of Agda, and explain the key differences between Agda and Haskell. Section 3 presents a naive translation of Haskell into Agda, and shows that it fails even for our simple queue example. In Section 4 we show how to solve this problem by introducing a monad of partiality—harder than it sounds in this setting. We present some sample proofs about translated programs in Section 5, to justify our claim that they are reasonably natural. Section 6 surveys related work, and finally Section 7 concludes and points out directions for future work.

2. An Overview of Agda

Agda is a proof assistant based on dependent type theory. Users construct a dependently typed functional program using an emacs interface which checks type correctness as the user works, and can also construct parts of the program automatically.
module Queue where

import Test.QuickCheck ((=>))

type Queue a = [a]
empty = []
add x q = q ++ [x]
isEmpty q = q == []
front (x:q) = x
remove (x:q) = q

type QueueI a = ([a],[a])
emptyI = ([],[])
addI x (f,b) = flipQ (f,x:b)
isEmptyI (f,b) = null f
frontI (x:f,b) = x
removeI (x:f,b) = flipQ (f,b)
flipQ ( [],b) = (reverse b, [])
flipQ q = q

retrieve :: QueueI a -> Queue a
retrieve (f,b) = f ++ reverse b

invariant :: QueueI Integer -> Bool
invariant (f,b) = null b || not (null f)

prop_empty =
retrieve emptyI == (empty :: [Integer])
prop_add x q =
retrieve (addI x q) == add x (retrieve q)
prop_isEmpty q =
invariant q =>
isEmptyI q == isEmpty (retrieve q)
prop_front q =
invariant q && not (isEmptyI q) =>
frontI q == front (retrieve q)
prop_remove q =
invariant q && not (isEmptyI q) =>
retrieve (removeI q) == remove (retrieve q)
prop_inv_empty = invariant emptyI
prop_inv_add x q =
invariant q => invariant (addI x q)
prop_inv_remove q =
invariant q && not (isEmptyI q) =>
invariant (removeI q)

Figure 2. The Queue Example in Haskell.

iterations include names as well as types for their fields. Recursive

types in Agda are interpreted inductively, so the type List a in-
cludes no partial or infinite lists.

Agda function definitions may also be recursive. For example, the
append function is defined as follows:

append :: (a::Set) -> List a -> List a -> List a =
\a |-> \xs -> \ys ->
case xs of (Nil) -> ys
(Cns x xs') -> Cns x (append xs' ys)

Polymorphic functions take explicit type arguments, although (as in this example) they can be “hidden”, indicated by the vertical bar “|−”. Hidden arguments can be omitted from calls, provided
Agda can infer what they should be, and this is often the case for
type arguments. This example also illustrates Agda’s dependent
types: the types of later arguments (xs and ys) and of the result
depend on the value of the first argument a. (This is a rather trivial
kind of dependent type, equivalent to a polymorphic one, because a
happens to be a Set, but Agda allows similar dependencies on
any type of argument).

Theorems and proofs are represented in Agda via the Curry-
Howard isomorphism: propositions are represented as types, whose
elements represent their proofs. Thus an empty type represents an
unprovable proposition (false), while a non-empty type represents a
provable one. Propositions are proved by constructing an element of
the corresponding type. For example, the polymorphic identity
function \(\text{a::Set} \rightarrow \text{\langle x:a \rangle \rightarrow x}\) proves the trivial propo-
sition \(A \Rightarrow A\), represented in Agda as the type \(\text{a::Set} \rightarrow a \Rightarrow a\).

In reasoning about programs, we often need to relate boolean
values in the code to Agda propositions, which are types. For this
reason, we define the type

\[ T :: \text{Bool} \rightarrow \text{Set} = \{ \text{b} \rightarrow \text{case b of (True )} \Rightarrow \text{Unit} \}
\]

which converts from one to the other. Thus, \(T \text{b}\) is a type which is
non-empty if and only if \(b\) is \text{True}. We shall illustrate the use of
this with a simple proof that if \(a\) and \(b\) is \text{True}, then so is \(a\).

We prove this by defining a function

\[
\text{lem1 :: (a,b::Bool) -> T (and a b) -> T a =}
\backslash a b \rightarrow \backslash pf \rightarrow \{!!\}
\]

which, for any booleans \(a\) and \(b\), given a proof that \(a\) and \(b\) is \text{True}, returns a proof that \(a\) is \text{True}. The \{!!\} on the right hand
side is a meta-variable which the emacs interface helps us to fill in.

One might expect to fill in the meta-variable with the value \(tt\),
since this is the only value that can be returned, but this would be
a type error: \(tt\) has the type \text{Unit}, and the type required here is
\(T \text{a}\), which might be either \text{Unit} or \text{Bottom} depending on the
value of \(a\). Instead we perform case analysis on \(a\). We enter \(a\) into
the meta-variable and issue a certain emacs command, whereupon
Agda inserts a case expression over the right type, with new meta-
variables in each branch:

\[
\text{lem2 :: (a,b::Bool) -> T (and a b) -> T a =}
\backslash a b \rightarrow \backslash pf \rightarrow \text{case a of (True )} \Rightarrow \{!!\}
\]

But now note that in each branch of the case, we know the
value of \(a\), and we can use this to simplify both the types of other
parameters, and the type needed as the result. For example, in the
\text{False} branch then \(pf\) has the type \(\text{T (and False b)}\), which
reduces to \text{T False} and thus to \text{Bottom}, and the type of the result
is \text{T False}, which also reduces to \text{Bottom}, so we can just return
\(pf\) in this case. The complete proof is:

\[
\text{lem2 :: (a,b::Bool) -> T (and a b) -> T a =}
\backslash a b \rightarrow \backslash pf \rightarrow \text{case a of (True )} \Rightarrow \{!!\}
\]

As demonstrated by this example, it is vital that Agda can use
the extra information gained by the case split for type-checking
the branches. To ensure this is always possible, Agda restricts case
expressions so that they may only inspect variables (in contrast to
Haskell cases which may inspect any expression); then the guarding
pattern of a branch (e.g., \text{False}) can be substituted for the subject
of the case (e.g., \(a\)). Moreover, case expressions may only appear
at the top level of a right-hand-side, i.e., as root expression of a
definition, function body, or case branch. Otherwise, one could
enter terms like

\[
t = \text{(case a of (True ) -> b)}
\]

which is morally equal to
correctness is valuable in itself. We shall adopt the same principle. It may seem odd, but it is a reasonable pragmatic decision because of the restrictions—which do not appear at the top-level of right-hand-sides, into local declarations. Haskell has also introduced a local function which is bound outside app1. The purpose of this function is simply to avoid polymorphic recursion (the type variable $\alpha$ is bound outside app1)—but the user of our translator would likely not expect it to appear. Although the translation to Core may appear complex and unpredictable, it does translate programs to a faithful representation of their semantics. Our thesis is thus that, provided proofs about programs depend only on the semantics of the translated terms, and not on their syntax, then the complexities of translation via Core will not cause complexity in the proofs themselves. We make the reasonable assumption that Haskell programmers conducting proofs understand the semantics of their code, and will not be surprised by the behaviour of the Core which GHC generates.

Apart from introducing explicit type abstractions and applications, putting type annotations on the binders, and translating pattern matching to a simple case, GHC has also introduced a local function app1. The purpose of this function is simply to avoid polymorphic recursion (the type variable $\alpha$ is bound outside app1)—but the user of our translator would likely not expect it to appear.

A few small differences between the syntax of Core and Agda require further processing. Because of the restrictions on case-expressions in Agda, we lift cases on non-variables, and cases which do not appear at the top-level of right-hand-sides, into local

3. A Naive Translation of Haskell to Agda

Haskell is a much more complex language than Agda, and contains many features that our translation must replace by simpler equivalents. Examples include list comprehension, do-notation, and nested and overlapping pattern matching. These can be interpreted as syntactic sugar, but must be desugared by our translator.

More awkwardly, Haskell programs are to a large extent implicitly typed, while Agda requires explicit typing, as we have seen. A translator must therefore infer types, and insert them into the translated code, together with type abstraction and application to represent polymorphic generalisation and instantiation. At the same time, overloading must be resolved, and overloaded definitions must be replaced by definitions parameterised on method dictionaries in the standard way [19]. Since Haskell’s class system has seen many extensions, this is far from trivial.

Fortunately, there is a tool already available which performs just such a translation—namely, the front-end of GHC. Internally, GHC translates Haskell programs to GHC Core, a simple language which is close to System F, with explicit typing, simple pattern matching, no overloading, and none of the other complex constructions alluded to above. A (slightly simplified) syntax of GHC Core appears in Figure 3.

Thus we begin our translation of Haskell to Agda by using GHC to translate the input to Core. This has the benefit of allowing us to work with a reasonably simple language, while at the same time, overloading must be resolved, and overloaded definitions must be replaced by definitions parameterised on method dictionaries in the standard way [19]. Since Haskell’s class system has seen many extensions, this is far from trivial.

A few small differences between the syntax of Core and Agda require further processing. Because of the restrictions on case-expressions in Agda, we lift cases on non-variables, and cases which do not appear at the top-level of right-hand-sides, into local

\[
\begin{align*}
\text{lem2} &:: (a,b)::\text{Bool} \rightarrow T \land (a \land b) \rightarrow T b = \\
&\quad \lambda a \rightarrow \lambda pf \rightarrow \text{case b of } (\text{True}) \rightarrow tt \text{ (False)} \rightarrow \{!!\}
\end{align*}
\]

the meta-variable cannot be filled with pf, because this has the type $T(\text{and} \; \text{False})$, which does not reduce to $T \; \text{False}$, and hence to Bottom, which is the type expected of the branch. The expression and a False is equal to False, but it does not reduce to it, which we can only see by inspecting the definition of and, which is given at the beginning of this section. This behaviour can catch novice users by surprise! On the one hand, building reduction into the Agda type-checker is very powerful—it shortens many proofs dramatically. On the other hand, it means the user must be very conscious of the difference between expressions which reduce to the same thing, and those which are merely provably equal (since proven equality cannot be exploited without an explicit proof step). The skillful Agda user needs to ensure that equalities needed in proofs are established, as far as possible, by pure reduction. This is important to bear in mind when planning a translation from Haskell.

Because Agda is intended as a proof editor, it is important that all expressions terminate—otherwise we could construct a proof of any proposition just by looping infinitely, in the same way we can use undefined in Haskell. Recursive definitions must therefore be total. This is not actually enforced by the Agda type-checker, which leaves the user to argue for termination separately. This may seem odd, but it is a reasonable pragmatic decision because of the difficulty of constructing good termination checkers which do not hinder expressivity too much, and because even proving partial correctness is valuable in itself. We shall adopt the same principle for our translation from Haskell to Agda. Haskell programs which loop infinitely will be translated into meaningless Agda models, and then all bets are off. Since we transfer this responsibility to the programmer, general recursive programs are not a problem for us to handle.

But forbidding infinite recursion is not enough to guarantee that evaluation always terminates. Agda makes two further restrictions—which are enforced by the system—namely, that case analysis is exhaustive, and that the type system is predicative. The latter restriction implies that it is not true that Set :: Set—which if allowed, would lead to Girard’s paradox, and thus non-termination. Rather, the type of Set is Type, and indeed there are an infinite number of nested “universes” ($\text{Set}, \text{Type}, \ldots$) in the Agda type system. Predicativity is not the only way to avoid Girard’s paradox, but it is the way adopted in Agda, partly for philosophical reasons. The immediate consequence is that polymorphic functions parameterised over types in Set cannot be instantiated at “larger” types such as Set itself. Both these restrictions are problematic for a translation from Haskell to Agda. We shall see how we deal with them in the following sections.

\[\text{case } a \; \text{of } (\text{True}) \rightarrow b \; d \; (\text{False}) \rightarrow c \; d\]
Our solution to this problem is to make definedness explicit in the Agda translation. We do so using the `Maybe` monad, so that the translations of defined expressions will have values of the form `Just v`, while undefined expressions, such as calls to `error`, will take the value `Nothing`. We are thus making partiality explicit in the translated definitions, enabling us to state and prove properties that involve partial values.

However, we do not want to commit ourselves to reasoning about partial values always. In many cases, partial values may be irrelevant, and we may wish to simplify proofs by restricting our attention to total elements. In other cases, the properties we wish to prove may simply be false for partial values—for example, that \( \text{reverse} \) is its own inverse—and we may wish to work in a setting where all values are total, rather than formulate and prove totality conditions at every turn. Luckily, we can have our cake and eat it too: we shall parameterize our translation on a monad \( m \), which we can take to be the `Maybe` monad when we reason about partiality, but the identity monad when we reason in a total setting. Thus our goal will be to develop a monadic translation of Core into Agda.

Our monad can be represented in Agda by three variables, which will be instantiated differently depending on the kind of reasoning we want to perform:

\[
m : \text{Set} \rightarrow \text{Set}
\]

\[
\text{return} : (\alpha : \text{Set}) \rightarrow \alpha \rightarrow m \alpha
\]

(\( \gg= \) : (\( \alpha : \text{Set} \)) \rightarrow (\( \beta : \text{Set} \)) \rightarrow m \alpha \rightarrow (\alpha \rightarrow m \beta) \rightarrow m \beta
\]

(Note that we hide the type parameters of \text{return} and \( \gg= \)). We can now apply the standard call-by-name monadic translation to the \( \lambda \)-calculus fragment of Core:

\[
\alpha^\dagger = m \alpha
\]

\[
(\tau_1 \rightarrow \tau_2)^\dagger = m(\tau_1^\dagger \rightarrow \tau_2^\dagger)
\]

\[
x^\dagger = x
\]

\[
(\lambda x.e)^\dagger = \text{return}(\lambda x.e)^\dagger
\]

\[
(e \ @ \ \tau)^\dagger = e^\dagger \gg= \lambda f.f^\dagger
\]

With this translation, function arguments are translated into monadic computations, which can thus be \text{Nothing} (undefined), correctly reflecting the lazy nature of Haskell. But there is a problem in translating type abstraction and application by this means. A natural approach is to translate type abstractions in the same way as \( \lambda \)-abstractions, so that

\[
(\forall \alpha. \tau)^\dagger = m((\alpha : \text{Set}) \rightarrow \tau^\dagger)
\]

\[
(\lambda \alpha.e)^\dagger = \text{return}(\lambda \alpha.e)^\dagger
\]

\[
(e \ @ \ \tau)^\dagger = e^\dagger \gg= \lambda f.f^\dagger
\]

This was the approach taken by Barthe, Hatter, and Thiemann (BHT) [1], but, for our purposes, it suffers two serious drawbacks.

The first drawback is that this translation does not correspond to "reality", that is, to the behaviour of Haskell implementations. Polymorphic values are here translated to \text{computations}, which may thus be undefined, but the result of instantiating them is also a computation, and may also be undefined. With this translation, when proving something about a function of type \( \forall \alpha. \alpha \rightarrow \alpha \), for example, we would have to first consider the case when the polymorphic value itself was undefined, and then separately consider the case when the polymorphic value was defined, but its instantiation at a particular type was undefined. In implementations of Haskell, these are the same value, so the distinction makes no sense—it would simply clutter every proof involving polymorphic values.

The second drawback is even more severe: the BHT translation, which is designed to translate from System F to itself, produces Agda terms which violate predicativity! Refer to the type of \( m \) again: it is \( \text{Set} \rightarrow \text{Set} \). But in the BHT translation, \( m \) is applied to the type \( (\alpha : \text{Set}) \rightarrow \tau^\dagger \)—which cannot be in \text{Set}, because it involves \text{Set} itself! Redefining \( m \) with type \( \text{Type} \rightarrow \text{Type} \) (where Type is the next universe beyond \text{Set}) does not help, because the monad \( m \) will also appear in the types which we instantiate \( \alpha \) to.

So if \( m \) were of type \( \text{Type} \rightarrow \text{Type} \), then we would need to abstract over Type instead of \text{Set} in the translation of polytypes, and so \( m \) would need to take an argument in the next universe instead... we would simply have pushed the problem one universe higher up. For
a predicative translation, we must avoid applying $m$ to polymorphic
types at all.

In fact, we do not need a monad to run into problems with
predicativity. System F, and thus Core, is already impredicative, and
permits terms such as $(\lambda \alpha . c)(\bar{\beta} \bar{\tau} \bar{\pi})$, which already instantiates a
type variable to a polytype. Luckily, the Core generated by GHC
rarely contains such examples\(^3\).

Since the impredicativity of System F causes problems, it is
natural to try to use a predicative fragment of it. The rank-1 (Hindley-
Milner) fragment is predicative, but too weak for Core, because
the translation of class dictionaries introduces higher-rank poly-
morphism in some cases\(^4\). However, even in these cases, we only
instantiate type variables to monomorphic types, and this is enough
to maintain predicativity (in fact, it is level 1 of Leivant’s Stratified
System F [14]). In practice, almost all Haskell programs are trans-
lated into Core within this fragment.

The basic idea behind our translation is to apply the monad only
to monomorphic types, that is, those whose translations are ele-
ments of $\text{Set}$ in Agda. Since the standard translation of function
types introduces an application of the monad on both the argument
and the result, for polymorphic functions, which will be represented
as functions with elements of $\text{Set}$ as arguments, we will need a dif-
ferent translation which does not involve the monad. We shall dis-
tinguish between the types of Core functions which are translated
monadically, and those which are not, by writing the latter in the
form $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$. Only the latter may have polytypes as arguments
or results. We use a preprocessor on the Core program to annotate
function types as either $\rightarrow$ or $\Rightarrow$, introducing the latter for functions
taking either types or class dictionaries (which may have polymor-
phic types) as parameters. Thus, the types in our annotated dialect
of Core are generated from the following grammar:

$$
\begin{align*}
\tau &::= \alpha \mid D \mid \tau \tau \mid \tau \rightarrow \tau \quad \text{monotypes} \\
\sigma &::= \tau \mid \forall \alpha : \kappa . \sigma \mid \sigma \rightarrow \sigma \quad \text{polytypes}
\end{align*}
$$

Since the translation of $\rightarrow$ functions is non-monadic, we can also
think of them as “unified” functions, for which we do not dis-
tinguish between non-termination of the function itself, and non-
termination of the calls. This fits well with the way we use them: when we reason about Haskell programs, we do not want to distin-
guish between non-termination before or after type instantiation, or
before or after dictionary passing.

Core quantifies not only over types, but type constructors of any
kind $\kappa ::= * \mid \kappa \rightarrow \kappa$. These are translated into Agda almost
literally:

$$
\begin{align*}
\kappa^1 &::= \text{Set} \\
\langle \kappa_1 \rightarrow \kappa_2 \rangle^1 &::= \kappa_1^1 \rightarrow \kappa_2^1 \\
\tau^1 &::= \text{m} \tau^*
\end{align*}
$$

Note that, for any kind $\kappa$, we have $\vdash \kappa^1 : \text{Type}$—its transla-
tion is an Agda Type. Now we can summarize the translation of Core
types:

$$
\begin{align*}
(\forall \alpha : \kappa . \sigma)^1 &::= \alpha^1 \rightarrow \sigma^1 \\
(\sigma_1 \rightarrow \sigma_2)^1 &::= \sigma_1^1 \rightarrow \sigma_2^1 \\
\tau^1 &::= \text{m} \tau^*
\end{align*}
$$

The last clause refers to the star translation $\tau^*$ of monotypes, which is homomorphic for variables, constants, and applications,
but applies the dagger translation on domain and codomain of function types:

$$
\begin{align*}
\alpha^* &= \alpha \\
D^* &= D \\
(\tau_1 \tau_2)^* &= \tau_1^* \tau_2^* \\
(\tau_1 \rightarrow \tau_2)^* &= \tau_1^* \rightarrow \tau_2^*
\end{align*}
$$

As expected, the translation of quantified types and $\rightarrow$ function
types is non-monadic.

The translation of Core $\lambda$-expressions and applications, and
type abstractions and instantiations, follows naturally from the
translation of types. We present the translation rules in Figure 4,
in the form of a translation from Core typing rules to valid
Agda typing derivations. Note that the translation of $\lambda$-abstractions
and applications depends on whether the function is of mono-
or polytype—functions with $\rightarrow$ types are translated into functions,
while functions of $\Rightarrow$ types are translated into monadic computa-
tions.

These rules also depend on a translation of environments:

$$
\begin{align*}
\{\}^1 &::= \{\} \\
(\Gamma ; \alpha : \kappa)^1 &::= \Gamma^1 ; \alpha : \kappa^1 \\
(\Gamma ; x : \sigma)^1 &::= \Gamma^1 ; x : \sigma^1
\end{align*}
$$

Note that the monad $m$ is only applied to elements of $\text{Set}$ in the
translated code!

Data-type definitions pose a special problem. They must be trans-
lated into Agda data-type definitions, but constructors in
Haskell (and thus in Core) are just functions with a type of the form
$\forall a : \kappa. \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow D \bar{a}$. The translation of such a
type is of the form $(\bar{a} : \kappa^1) \rightarrow (\tau_1^1 \rightarrow \cdots \tau_n^1 \rightarrow m (\text{D} \bar{a}))$, and
of course, no Agda constructor can have such a type. There-
fore, Haskell constructors cannot be translated directly into Agda
constructors. Instead, we will introduce Agda constructors with
types of the form $(\bar{a} : \kappa^1) \rightarrow \tau_1^1 \rightarrow \cdots \tau_n^1 \rightarrow D \bar{a}$, whose
components have monadic types, but whose type is not otherwise
monadic, reflecting the fact that the result of a constructor is never
undefined.

This is formalised as follows. The typing rules in both Core and
Agda for data declarations involve judgements of the forms:

$$
\Gamma \vdash d : \Delta \\
\Gamma ; \bar{a} : \kappa ; D : \kappa \rightarrow * \vdash cd : \Delta
$$

These judgements infer the names and types that will be added to
the context as the result of the declaration—that is, the names and
types of the constructors declared. The translation of these
judgements from Core to Agda is given in Figure 5. The first
rule translates data declarations in Core to declarations in Agda
of a type with the same name, by translating each constructor
and collecting the results. The second rule specifies the translation
of constructor types. The translation includes fresh field-names,
because Agda syntax demands them.

Fortunately, Core permits only full applications of constructors\(^5\)
to all of their arguments—partial applications in Haskell are $\eta-
converted to $\lambda$-expressions during translation to Core. Thus we
need only translate full applications to Agda:

$$
\Gamma \vdash C @ \bar{e} : D \bar{e}^1 \vdash \text{return} (C \bar{e}^1) : m (D \bar{e}^1)
$$

Note that the type arguments of a constructor are implicit in Agda.

\(^5\) Contrary to Haskell, Core constructors cannot have strict fields—if a
constructor is strict in an argument in Haskell, that argument is evaluated
explicidy before the constructor is applied in Core.
The translation of case expressions appears in Figure 6. Clauses alt are typed using a four-ary judgment $\Gamma \vdash \text{alt} : \tau_1 \Rightarrow \tau_2$, where $\tau_1$ is the type of the pattern and $\tau_2$ the type of the branch. We use monadic $\Rightarrow$ to evaluate the expression the case inspects, so the actual Agda case analysis of the value appears in the second argument of $\Rightarrow$. As we explained in Section 2, Agda case expressions are syntactically restricted to appear at the top level of a definition, and so we “lambda-lift” the second argument of $\Rightarrow$ to a freshly named locally defined function to fulfill this restriction.

### 4.1 An Optimisation

If the rules above are applied literally, they generate Agda definitions with a very large number of monadic operations. As an example, consider a function defined as

\[ f : \alpha \rightarrow \beta \]

\[ f = \lambda (x : \alpha). e \]

The translation of this definition is

\[ f : m (\alpha \rightarrow \beta) \]

\[ f = \text{return } \lambda (x : \alpha). e \]

Note that $f$ itself is assigned a monadic type, and that applications $f e$ are translated to $f \Rightarrow e$. Functions of many arguments are assigned even more complex types, and become even more cumbersome to invoke.

This is a problem, because the Agda user writes proofs and properties which refer to translated definitions. If just invoking a translated function is cumbersome, then these proofs will be even more cumbersome. Fortunately, there is a simple solution to this problem.

Within the scope of the definition above, we know that $f$ cannot be $\bot$, and so assigning it a monadic type and invoking it via $\Rightarrow$ is just overkill. Our translator therefore omits the application of the monad to the $\lambda$-expression, generating the optimised translation

\[ f : \alpha \rightarrow \beta \]

\[ f = \lambda (x : \alpha). e \]

instead. We restrict the applications of $f$ to be full applications (\textit{r}-converting if need be), which can then be translated simply via $(f e)^* = f e$. In effect, we treat defined functions in the same
way as constructors, and the only complication is that our translator must keep track of the arity of such definitions. The benefit is that the Agda user can then invoke translated functions from proofs via ordinary Agda function application, as though these functions had been defined in Agda in a normal way themselves.

5. Case Study: Sample Proofs
In this section we present sample proofs as they were type-checked by the Agda proof-assistant. We also present sample results of our monadic translation, but here we have renamed variables and adjusted layout for readability. Since the user performing proofs normally need only refer to the Haskell source code, and not its translation, then the actual variable names and layout in the translated code are irrelevant.

5.1 Case 1: Lists
Our first example is the monadic translation of lists and the `append` function on lists, as defined in the Haskell Prelude. The result of translation is as follows:

```haskell
data List (a :: Set) = Nil | Cons (mx :: m a) (mxs :: m (List a))

(++) :: (a :: Set) |\rightarrow m (List a) \rightarrow m (List a) \rightarrow m (List a)
= \lambda a \rightarrow
  let appl :: m (List a) \rightarrow m (List a) \rightarrow m (List a)
  = \lambda mxs mys \rightarrow
    let app2 :: (xs :: List a) :: m (List a)
    case xs of
      (Nil) \rightarrow mys
      (Cons x xs') \rightarrow
        return (Cons x (appl xs' mys))
    in mxs >>= app2
  in appl
```

(where the list type and constructors are renamed to conform to Agda’s syntax).

Notice that now, the arguments of `Cons` are of type `m a` and `m (List a)` instead of `a` and `List a`, respectively. Similarly, the arguments of `(++)` also have monadic types. Hence, in the definition of the functions we use `return` and `(>>=)` for returning elements or applying functions to arguments, respectively. Despite the type information, the `let` expressions and the explicit `case` analyses, we believe the translated code is easy to follow.

In this example, we prove the associativity of append both in the identity (`Id`) monad and in the `Maybe` monad.

Identity Monad
When doing the proofs in the `Id` monad we instantiate `m`, `return` and `(>>=)` as follows:

```haskell
m :: Set \rightarrow Set = \lambda a \rightarrow a
return :: (a::Set) |\rightarrow a \rightarrow m a = \lambda a \rightarrow \lambda x \rightarrow x
(>>=) :: (a,b::Set) |\rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b
= \lambda a b \rightarrow \lambda ma \rightarrow \lambda mb \rightarrow ma >>= mb
```

We can now define an equality relation over lists on the `Id` monad as follows:

```haskell
Eq :: (a::Set) |\rightarrow (eq::a \rightarrow a \rightarrow Set) \rightarrow
List a \rightarrow List a \rightarrow Set
= \lambda a \rightarrow \lambda eq \rightarrow \lambda xs ys
  case xs of
    (Nil) \rightarrow case ys of
      (Nil) \rightarrow Unit
      (Cons ma' mas') \rightarrow Bottom
      (Cons ma mas') ->
    case ys of
      (Nil) \rightarrow Bottom
      (Cons ma' mas') \rightarrow And (eq ma ma')
      (Eq eq mas mas')
```

Here, `And` is the conjunction of sets defined as data `And (a,b::Set) = and (x::a) (y::b). If the equality on the set `a` is reflexive, then we can prove that equality of lists is also reflexive.

```haskell
reflEqList :: (a::Set) |\rightarrow (eq::a \rightarrow a \rightarrow Set) \rightarrow
  (refl1::(x::a) \rightarrow eq x x) \rightarrow
  (xs::List a) \rightarrow Eq eq xs xs
```

Let us assume we have a set `s` and a reflexive equality relation over `s`:

```haskell
EqS :: s \rightarrow s \rightarrow Set
reflS :: (x :: s) \rightarrow EqS x x
```

Then the associativity of append can now be proved as follows:

```haskell
app_assoc :: (xs,ys,zs::List s) \rightarrow
EqEq (xs ++ (ys ++ zs))
  ((xs ++ ys) ++ zs)
= \lambda xs ys zs \rightarrow
case xs of
  (Nil) \rightarrow reflEqList EqS reflS (ys ++ zs)
  (Cons x xs') \rightarrow and (reflS x)
  (app_assoc xs' ys zs)
```

When the first argument of append is empty, the definition of append simple reduces the second argument. Hence, when `xs` is empty the property amounts to proving that `ys` of empty lists is equal to itself, which is true by the reflexivity of the equality on lists. On the other hand, if `xs` is not of the form `Cons x xs'`, by definition of append we need to prove that `Cons x (xs' ++ (ys ++ zs))` is equal to `Cons x ((xs' ++ ys) ++ zs)`, or equivalently, to `Cons x ((xs' ++ y) ++ zs)`, again by definition of append. For both lists to be equal we need to provide a proof that the first elements in both lists are equal, which is provided by `reflS x`, and a proof that the rests of the lists are also equal, which is provided by the inductive hypothesis.

This proof is the same as it would be if no monads were involved, which is not surprising since we are working with the `Id` monad, whose operations reduce away entirely.

Maybe Monad
When working in the `Maybe` monad we instantiate `m`, `return` and `(>>=)` as follows:

```haskell
m :: Set \rightarrow Set = \lambda a \rightarrow Maybe a
return :: (a::Set) |\rightarrow a \rightarrow m a
  = \lambda a \rightarrow \lambda x \rightarrow Just x
(>>=) :: (a,b::Set) |\rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b
  = \lambda a b \rightarrow \lambda ma \rightarrow \lambda mb \rightarrow case ma of
    Nothing \rightarrow Nothing
    (Just x) \rightarrow mb
```

As before, we assume a set `s` with an equality relation `EqS` that is reflexive (`reflS`), and we then define a reflexive equality relation over `Maybe` lists which we again call `Eq`. Notice that now the result of append is a `Maybe` list, thus we should also define an equality relation over the `Maybe` type. If the equality over the argument set of `Maybe` is reflexive, then we can prove that the equality over the `Maybe` type is also reflexive (the type of this statement is presented below).

```haskell
EqM :: (a::Set) |\rightarrow (eq::a \rightarrow a \rightarrow Set) \rightarrow
(m1,m2:: Maybe a) \rightarrow Set
= \lambda a \rightarrow \lambda eq \rightarrow \lambda m1 m2 ->
```

```haskell
Case Study: Sample Proofs
In this section we present sample proofs as they were type-checked by the Agda proof-assistant. We also present sample results of our monadic translation, but here we have renamed variables and adjusted layout for readability. Since the user performing proofs normally need only refer to the Haskell source code, and not its translation, then the actual variable names and layout in the translated code are irrelevant.

5. Case Study: Sample Proofs
In this section we present sample proofs as they were type-checked by the Agda proof-assistant. We also present sample results of our monadic translation, but here we have renamed variables and adjusted layout for readability. Since the user performing proofs normally need only refer to the Haskell source code, and not its translation, then the actual variable names and layout in the translated code are irrelevant.

5.1 Case 1: Lists
Our first example is the monadic translation of lists and the `append` function on lists, as defined in the Haskell Prelude. The result of translation is as follows:

```haskell
data List (a :: Set) = Nil | Cons (mx :: m a) (mxs :: m (List a))

(++) :: (a :: Set) |\rightarrow m (List a) \rightarrow m (List a) \rightarrow m (List a)
= \lambda a \rightarrow
  let appl :: m (List a) \rightarrow m (List a) \rightarrow m (List a)
  = \lambda mxs mys \rightarrow
    let app2 :: (xs :: List a) :: m (List a)
    case xs of
      (Nil) \rightarrow mys
      (Cons x xs') \rightarrow
        return (Cons x (appl xs' mys))
    in mxs >>= app2
  in appl
```

(where the list type and constructors are renamed to conform to Agda’s syntax).

Notice that now, the arguments of `Cons` are of type `m a` and `m (List a)` instead of `a` and `List a`, respectively. Similarly, the arguments of `(++)` also have monadic types. Hence, in the definition of the functions we use `return` and `(>>=)` for returning elements or applying functions to arguments, respectively. Despite the type information, the `let` expressions and the explicit `case` analyses, we believe the translated code is easy to follow.

In this example, we prove the associativity of append both in the identity (`Id`) monad and in the `Maybe` monad.

Identity Monad
When doing the proofs in the `Id` monad we instantiate `m`, `return` and `(>>=)` as follows:

```haskell
m :: Set \rightarrow Set = \lambda a \rightarrow a
return :: (a::Set) |\rightarrow a \rightarrow m a
  = \lambda a \rightarrow \lambda x \rightarrow Just x
(>>=) :: (a,b::Set) |\rightarrow m a \rightarrow (a \rightarrow m b) \rightarrow m b
  = \lambda a b \rightarrow \lambda ma \rightarrow \lambda mb \rightarrow case ma of
    Nothing \rightarrow Nothing
    (Just x) \rightarrow mb
```

As before, we assume a set `s` with an equality relation `EqS` that is reflexive (`reflS`), and we then define a reflexive equality relation over `Maybe` lists which we again call `Eq`. Notice that now the result of append is a `Maybe` list, thus we should also define an equality relation over the `Maybe` type. If the equality over the argument set of `Maybe` is reflexive, then we can prove that the equality over the `Maybe` type is also reflexive (the type of this statement is presented below).

```haskell
EqM :: (a::Set) |\rightarrow (eq::a \rightarrow a \rightarrow Set) \rightarrow
(m1,m2:: Maybe a) \rightarrow Set
= \lambda a \rightarrow \lambda eq \rightarrow \lambda m1 m2 ->
```

```haskell
```
case m1 of
  (Nothing) -> case m2 of (Nothing) -> Unit
                (Just x) -> Bottom
  (Just x) -> case m2 of (Nothing) -> Bottom
               (Just x') -> eq x x'

ref1EqM :: (a::Set) |-> (eq::a -> a -> Set) ->
           (ref1 :: (x::a) -> eq x x) ->
           (ma :: Maybe a) -> EqM eq ma ma

The desired property can now be proved as follows:

app_assoc :: (mxs,mys,mzs::m (List s)) ->
EqM (Eq EqS) (mxs ++ (mzs ++ mys)))
      ((mzs ++ mys) ++ mzs)
= \mxs \mzs \mys ->
case \mzs of
  (Nothing) -> tt
  (Just x) ->
case x of
    (Nothing) -> ref1EqM (Eq EqS)
      (ref1EqLst EqS ref1S)
      ((mzs ++ mys))
      (Cons ma mzs) ->
      and (ref1EqM EqS ref1S ma)
      (app_assoc mas mys mzs)

Let us once again analyse this property. Remember that we are now working with Maybe lists, so when we do case analysis on the list \mzs we obtain the cases Nothing and Just \xs. When \mzs is Nothing we simply need to prove that Nothing is equal to itself in the equality relation over the Maybe type, which is trivial. If, on the other hand, \mzs is the list \xs, we perform case analysis on the list. The proof now continues in a similar way to the one for the Id monad, only that now we need a proof on the equality relation over the Maybe type and not on the equality relation over lists.

Other Monads

The reader may well wonder why we prove the same property twice, for two different monads—why not just prove it once-and-for-all, for any monad? While this may seem attractive in principle, it turns out to be much more difficult in practice.

When we instantiate the monad parameters with a specific monad and its operations, and we perform case analyses on monadic terms, then Agda is able to perform reduction steps in the type of the property we wish to prove and, in this way, simplify such a type. This simplification allows us to provide concrete terms that prove the property for each of the cases in the case analyses. On the other hand, when we attempt proofs about a general monad, the only thing the proof assistant knows is the types of the properties to be proved. Since Agda does not know how to compute with a general monad it will not be able to simplify the type of the properties by performing reduction steps. Thus, the only thing we can do to prove properties is to use the monad laws explicitly. Although it is possible to prove properties in this way, those proofs are both more difficult to perform and to understand, and much longer than the ones we presented above.

5.2 Case 2: Queues

We discuss here the monadic translation of the queue example presented in Figure 2. The monadic translation of the datatype of list has already been presented. Below we show the translation of Boolean values and pairs.

data Bool = False | True

data Pair (a,b::Set) = P (ma :: m a) (mb :: m b)

The translations of most of the functions in Figure 2 are similar to the one of (++) discussed in the previous section. For example, the add operation on queues implemented as lists is defined in type theory as follows:

add :: (a::Set) |-> m a -> m (List a) -> m (List a)
    = \a |-> \mxs -> \mzs ->
       mqs ++ (return (Cons mx (return Nil)))

One a thing we should point out is that, since we take the GHC Core code as the starting point of our translator and since GHC sometimes inlines function applications, the translations of some functions are syntactically, not exactly as we expect them to be (although they are semantically equivalent to their expected versions). In addition, GHC replaces type definitions with the defined types. The translation of the function \texttt{add} exemplifies these two points.

Here, GHC has inlined the application to the function \texttt{flip1} and replaced the type of queues by the type of pairs of lists.

add::(a::Set) |-> m a ->
    m (Pair (List a) (List a)) ->
    m (Pair (List a) (List a))
    = \a |-> \ma |-> \mp |->
      let add11 (p :: Pair (List a) (List a))
       :: m (Pair (List a) (List a))
       = case p of
          (P f b) ->
          let add2 (xs :: List a)
          :: m (Pair (List a) (List a))
          = case xs of
             (Nil) ->
             return (P (reverse
                      (return (Cons ma b))))
             (return Nil)
             (Cons mx mzs) ->
             return (P (return xs)
                       (return (Cons ma b)))
          in f >>= add2

However, the translations of the functions returning the first element of a non-empty queue (\texttt{front} and \texttt{front1}) or the remaining part of a non-empty queue after removing its first element (\texttt{remove} and \texttt{remove1}) do not follow the same schema as the other functions. The reason is that these functions are not defined for all queues but only for non-empty queues. In Haskell this is done by simply leaving out some cases when defining the functions by pattern matching. However, in type theory, each function must be total, which means we must also define the functions front and remove for empty queues. The translations of these functions are thus only defined for the Maybe monad, and we make use of the value Nothing when attempting an application of any of these functions to the empty queue. Below we present the translation of the function remove. The other functions are translated as expected in the Maybe monad, except for the inlining of the function \texttt{flip1}.

remove :: (a :: Set) |-> m (List a) -> m (List a)
    = \a |-> \mxs ->
      let rem (xs :: List a) :: m (List a)
          = case xs of
              (Nil) -> Nothing
              (Cons mx mxs') -> mxs'
      in mxs >>= rem

Maybe Monad

Since some of the functions are only defined for the Maybe monad, we have performed all the proofs in the Maybe monad. Those proofs not involving the partial functions can, of
course, also be performed in the identity monad. For the sake of
readability, let us reintroduce the definition of queues before we
continue.

Queue (a::Set) :: Set = List a
Queue1 (a::Set) :: Set = Pair (List a) (List a)

In formulating properties, we chose not to use the monadic
translation of the Haskell invariant as the invariant in our proofs.
(Recall that the Haskell invariant was:)

invariant :: Queue1 Integer -> Bool
invariant (f,b) = null b || not (null f)

Rather, we reformulated the invariant directly in type-theory. The
reasons for this were as follows.

First, the Haskell invariant was originally defined for testing
the queue properties with QuickCheck. Since QuickCheck cannot
handle polymorphic properties, the invariant was instantiated to
Integer queues. But we want to reason about all queues.

Then, QuickCheck generates only total lists, or pairs of total
lists, when checking the properties—but totality is not represented
in the Haskell invariant. A property that is true for total queues
might not be true for non-total queues. Since boolean disjunction
is not strict in Haskell and True || undefined evaluates to True,
the Haskell invariant is true for the queue (undefined,[]).

Adding an element to this queue results in the queue undefined
which of course violates the invariant. Consequently, property
prop_inv_add, which states that adding an element preserves the
invariant, fails for partial queues. Hence, we need to make the
totality of the lists in a queue an explicit part of the invariant when
we work in a partial setting.

A third reason is that, while in Haskell it is enough to know
whether a property is true or false, in type theory we need more
information: we need a proof of its truth or falsity. When one of
the premises in a property is true for a certain input, we might
need to manipulate the concrete evidence of that truth. Hence, if
we define the invariant as a complex boolean expression and we
simply translate the invariant by lifting the True and False values
into the true set Unit or the false (empty) set Bottom, respectively,
we might lose information. Instead, we define the type-theoretic
invariant by lifting every single piece of Boolean information, and
then we manipulate the resulting sets in type theory with logical
operators on sets.

Thus, the invariant we define is the following:

invariant :: (a :: Set) |-> m (QueueI a) -> Set
= \a |-> \mp -> undefined
    let mf = mfst mp; mb = msnd mp
    in And3 (TM (totalList mf))
    (TM (totalList mb))
    (Or (TM (null mb)) (TM (not (null mf))))

Here, mfst and msnd select the first and second element, respectiv-
ely, of a Maybe pair, And3 is defined similarly to And but it per-
forms the conjunction of three sets instead of two, and Or is the
disjunction of sets defined as

data Or (a,b::Set) = inl (x::a) | inr (y::b)

Finally, TM lifts a Maybe Bool into a set. Its definition is similar to
that of T in Section 2, except that it also lifts the value Nothing to
the set Bottom.

As before, we assume a set s with a reflexive equality relation.
Two of the properties we wish to prove are trivial.

prop_empty :: EqM (Eq EqS)
(retrieve (emptyI s s))
(empty s) =
reflEqM (Eq EqS) (reflEqLst EqS reflS) (empty s)
prop_inv_empty :: invariant (emptyI s s)
= and3 tt tt (inl tt)

The remaining six properties are also rather simple when we
work with the invariant we defined above. The structures of the
proofs are similar in all the proofs: we perform a few case analyses
and we distinguish the cases where the input is partial and the cases
were the input is total. In this example, the cases of the first type are
easily eliminated by absurdity. For the cases of the second type, we
need to provide concrete proofs of the statement we wish to prove.
Let us study two of the remaining proofs here.

The first example is the proof of the property prop_add.

prop_add :: (ma :: m =: m (QueueI s)) -> EqM (Eq EqS) (retrieve (addI ma mq))
= \ma -> \mq ->
case mq of
    (Nothing)-> tt
    (Just q )->
        case q of
            (P mf mb)->
            case mf of
                (Nothing)-> tt
                (Just f )->
                    case f of
                        (Nil)-> app_nil (reverse
                            (Cons (Cons ma mb)))
                        (Cons mx msx)->
                            app_assoc mf (reverse mb)
                            (Just (Cons ma (Just Nil)))

When the queue is Nothing or when its front list is Nothing
the property is trivial since it amounts to proving that Nothing
is equal to itself. Otherwise, we need to distinguish whether the front
list is empty or not. Here, appnil msx is a proof that mxs ++
(return Nil) is equal to mxs, formed by mxs :: m (List s).

Finally we consider a property on the function removeI.

prop_inv_removeI :: (mp :: m (QueueI s)) ->
invariant mp -> TM (not (isEmptyI mp)) ->
invariant (removeI mp)
= \(mp::m (QueueI s))->
\(!\(\text{invariant}\ mp))->
\(!\(\text{not} \text{(isemptyI \ mp)}))->
case inv of
    (and3 tf tb nl)->
    case mp of
        (Nothing)-> case tf of { }
        (Just p )->
            case p of
                (P mf mb)->
                case mf of
                    (Nothing)-> case tf of { }
                    (Just x )-
                        case xs of
                            (Nil)-> case ne of { }
                            (Cons mx msx)->
                                case msx of
                                    (Nothing)-> case tf of { }
                                    (Just x ')->
                                        case x ' of
                                            (Nil)->
                                            and3 (tot2rev_tot mb tb)
                                            tf (inl tt)
                                            (Cons mx' msx')-
                                            and3 tf tb (inr tt)
Since the invariant of \( m \) is true, \( m \) cannot be \texttt{Nothing}, and neither can it contain a sub-part that is \texttt{Nothing}. In addition, its front list cannot be empty, since this would contradict the third hypothesis. Once we have discarded the absurd cases, we need to prove the invariant of \( \text{Just} \ (P \ (\text{Just} \ (\text{Cons} \ x \ (\text{Just} \ x'))) \ mb \), for the cases where \( x' \) is empty and it is not empty. Both cases are easy. Here, \texttt{tot2rev\_core} is a proof that the reverse of a total list is a total list.

Although there are many case analyses in these proofs, recall that they are easy to construct: it is only necessary to tell Agda on which variable we would like to perform the analysis, and Agda then produces all the cases we need for that particular expression, leaving us with a goal to fill in for each of the cases we need to consider.

In order to fill-in each of these goals, it was enough to understand how the Haskell definitions work and what the results of the functions were when we applied them to a partial list or queue (of the form \texttt{Nothing}). We did not need to inspect the translated definitions. In this sense, it did not really matter that GHC inlined some of the function applications, or that the indentation or names in the codes resulting from our translator could be improved. This had no consequence whatsoever when proving the properties.

The inlining of function applications might have consequences, though, when we need to relate a property of the inlined function with a property of the functions that use the inlined function—since now the later function does not refer explicitly to the former one. But in this case, we can switch off inlining with the GHC pragma \texttt{NOLINING}.6

6Desirable would be a flag to GHC which turns off all inlining in the translation to Core.

6. Related Work

The monadic translation of Barthe, Hatchiff, and Thiennam [1] has been discussed in Section 4. Uustalu [18] presents a monadic translation of inductive and coinductive simple types with iteration and coiteration schemes. He encodes data types via binary products, bi-iteration of inductive and coinductive simple types with iteration and

coiteration schemes. He encodes data types via binary products, bi-

...
in the Maybe monad, and all proofs that involve it must take partiality into account. We would like to be able to refer to such functions in proofs about total elements, when we know that their preconditions are satisfied. While it is straightforward to map partial values, and a proof of their totality, back to total values, we have not yet found a way to do so that does not clutter proofs unacceptably.

Finally, Capretta [3] demonstrates that general recursion can also be captured in a monad, using a coinductive type. It would be interesting to instantiate our translations with this monad too, although this would require extending Agda with co-inductive types.

In summary, we have presented a workable way to prove Haskell programs correct in type-theory based provers.

Acknowledgments
We thank the members of the CoV er project: Thierry and Cat erina Coquand for theoretical and practical development of Agda according to our needs; Gregoire Hamon for the implementation of the majority of CoverTranslator, in which we could plug-in our monadic translation; Patrick Jansson and Nils Anders Danielsson for a preliminary implementation of the naive translation; Fredrik Lindblad for his development of Agsy; and Koen Claessen, Peter Dybjer, and Mary Sheeran for discussions on proofs about partial programs. We are indebted to the anonymous referees which took aside some of their valuable time to carefully read the draft version of this paper and provided helpful comments to improve it.

References