Verifying Haskell Programs Using Constructive Type Theory *

Andreas Abel Marcin Benke Ana Bove John Hughes Ulf Norell

Chalmers University of Technology

{abel,marcin,bove,rjmh,ulfn}@cs.chalmers.se

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,

$$\forall \text{xs} :: \text{List Integer}.\exists \text{ys} :: \text{List Integer}.\\ \text{isPermutation}(\text{xs}, \text{ys}) \wedge \text{isOrdered}(\text{ys})$$

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

* This work has been funded by the Swedish Foundation for Strategic Research (SSF) through the project *CoVer*. The first three authors were additionally supported by the coordination action *TYPES* (510996) of the European Union, and the first, second, and fourth author by the EU thematic network *Applied Semantics II* (IST-2001-38957).

Haskell'05 September 30, 2005, Tallinn, Estonia.

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.

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.

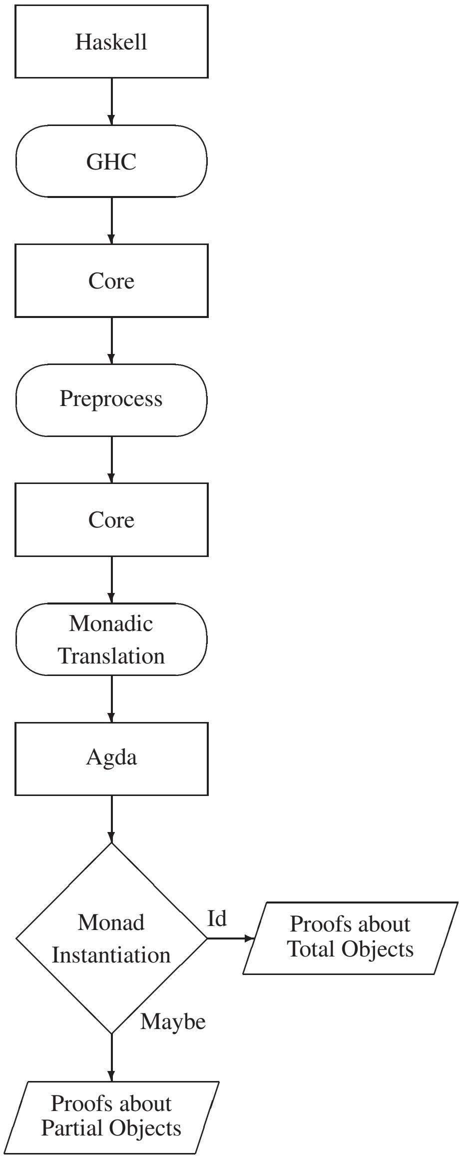

Figure 1. Translation Outline.

Agda proofs consist of a collection of explicitly typed definitions, such as

```
and :: Bool -> Bool -> Bool =
  \a -> \b -> case a of (True )-> b
                        (False)-> a
```

New data types can be defined with a Haskell like syntax; for example

```
data Unit = tt
data Bottom =
```

define the one-point type and the empty type respectively. Data type definitions can be parameterised and recursive, as in the type of lists:

```
data Lst(a::Set) = Nl | Cns (x::a) (xs::Lst a)
```

Types such as these are first-class values, of the type `Set`, which is thus the type of the parameter to `Lst`. Note that constructor dec-

```
module Queue where

import Test.QuickCheck

type Queue a = [a]
empty        = []
add x q      = q ++ [x]
isEmpty q    = null 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::Integer) 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.

larations include names as well as types for their fields. Recursive types in Agda are interpreted *inductively*, so the type `Lst a` includes no partial or infinite lists.

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

```
append :: (a::Set) |-> Lst a -> Lst a -> Lst a =
  \a |-> \xs -> \ys ->
    case xs of (Nl       )-> 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 in `|->`. 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 `\(a::Set) -> \(x::a) -> x` proves the trivial proposition $A \Rightarrow A$, represented in Agda as the type `(a::Set)->a->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 :: Bool -> Set = \b -> case b of (True )-> Unit
                                   (False)-> Bottom
```

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

We prove this by defining a function

```
lem1 :: (a,b::Bool) -> T (and a b) -> T a =
   \a b ->  \pf -> {!!}
```

which, for any booleans `a` and `b`, given a proof that `and a b` is `True`, returns a proof that `a` is `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 `Unit`, and the type required here is `T a`, which might be either `Unit` or `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:

```
lem1 :: (a,b::Bool) -> T (and a b) -> T a =
   \a b ->  \pf -> case a of (True )-> {!!}
                             (False)-> {!!}
```

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 `False` branch then `pf` has the type `T(and False b)`, which reduces to `T False` and thus to `Bottom`, and the type of the result is `T False`, which also reduces to `Bottom`, so we can just return `pf` in this case. The complete proof is:

```
lem1 :: (a,b::Bool) -> T (and a b) -> T a =
   \a b ->  \pf -> case a of (True )-> tt
                             (False)-> pf
```

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., `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 = (case a of (True ) -> b
               (False) -> c) d
```

which is morally equal to

```
case a of (True ) -> b d
          (False) -> c d
```

however, not w. r. t. β-reduction, but by virtue of a so-called *permutation*. To avoid permutations and the additional complications to type-checking which terms like `t` provoke, such terms are forbidden in Agda. The two restrictions on `case` complicate the translation of Haskell to Agda somewhat.

Agda accepts that two types match *if they reduce to the same term*, so reduction is of critical importance in formulating Agda proofs. For example, if we tried to prove that `T(and a b) -> T b` instead, by case analysis on the variable `b`, then type checking would fail. In the partial proof

```
lem2 :: (a,b::Bool) -> T (and a b) -> T b =
   \a b ->  \pf -> case b of (True )-> tt
                             (False)-> {!!}
```

the meta-variable *cannot* be filled with `pf`, because this has the type `T(and a False)`, which does not *reduce* to `T 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" (`Set`, `Type`, ...) 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.

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 a far from trivial task.

Fortunately, there is *already* a tool 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 supporting full Haskell. However, GHC does change the program structure in ways one might not expect, which may complicate the reasoning later. For example, the Core translation of the `++` function

```
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

is (in mathematical notation, we write the typing relation as ":")

$$
\begin{array}{lll}
(+\!\!+) & : & \forall\alpha : *.\ \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha \\
 & = & \Lambda(\alpha : *). \\
 & & \quad \textbf{let}\ \ app1 : \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha \to \mathsf{List}\ \alpha \\
 & & \qquad\quad = \ \lambda(ds : \mathsf{List}\ \alpha).\ \lambda(ys : \mathsf{List}\ \alpha). \\
 & & \qquad\qquad \textbf{case}\ ds\ \textbf{of} \\
 & & \qquad\qquad\quad \mathsf{Nil} \to ys \\
 & & \qquad\qquad\quad \mathsf{Cons}\ (x : \alpha)\ (xs : \mathsf{List}\ \alpha) \to \\
 & & \qquad\qquad\qquad \mathsf{Cons}\ @\ \alpha\ x\ (app1\ xs\ ys) \\
 & & \quad \textbf{in}\ \ app1
\end{array}
$$

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 $app1$[1]. The purpose of this function is simply to avoid polymorphic recursion (the type variable α 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.

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 **case**s on non-variables, and **case**s which do not appear at the top-level of right-hand-sides, into local

[1] Actually it is called $+\!\!+1$; we have taken the liberty of renaming it.

$$
\begin{array}{lcl}
d & ::= & \mathbf{data}\, D\, (\vec{\alpha} : \vec{\kappa}) = cd_1 \mid \ldots \mid cd_n \\
 & \mid & x : \sigma = e \\
cd & ::= & C\, (\vec{\alpha} : \vec{\kappa})\, \vec{\sigma} \\
e & ::= & x \mid C \mid e\, e \mid \lambda(x : \sigma).\, e \\
 & \mid & e\, @\, \tau \mid \Lambda(\alpha : \kappa).\, e \\
 & \mid & \mathbf{let}\, x_1 : \sigma_1 = e_1; \ldots ; x_n : \sigma_n = e_n\, \mathbf{in}\, e \\
 & \mid & \mathbf{case}\, e\, \mathbf{of}\, alt_1; \ldots ; alt_n \\
alt & ::= & C\, (\vec{x} : \vec{\sigma}) \to e \\
\sigma & ::= & \tau \mid \forall \alpha : \kappa. \sigma \mid \sigma \mapsto \sigma \\
\tau & ::= & \alpha \mid D \mid \tau\, \tau \mid \tau \to \tau \\
\kappa & ::= & * \mid \kappa \to \kappa
\end{array}
$$

Figure 3. A simplified grammar for GHC Core

definitions as follows:

$$
\begin{array}{c}
\mathbf{case}\, e\, \mathbf{of}\, alts \\
\Downarrow \\
\mathbf{let}\, f\, x = \mathbf{case}\, x\, \mathbf{of}\, alts\, \mathbf{in}\, f\, e
\end{array}
$$

Moreover, Core **case**-expressions may contain default cases abbreviating all remaining constructors—we simply expand these to the constructor cases they represent. We also translate type abstraction and application using Agda's hidden parameters, so that the final Agda translation of the `append` function becomes

```
(++) :: (a :: Set) |-> List a -> List a -> List a
  = \a |->
    let app1 :: List a -> List a -> List a
          = \ds ys ->
              case ds of
                (Nil      ) -> ys
                (Cons x xs) -> Cons x (app1 xs ys)
    in app1
```

Clearly, this translation doesn't take the question of termination into account; if we translate a Haskell program with an infinite loop into Agda in this way, we will obtain a meaningless Agda program. But the problem is actually much more immediate: the translation fails even for the simple `Queue` example in Figure 2, which refers only to structurally recursive functions! The problem is that `front` and `remove` are partial functions—not because they may loop infinitely, but because they do not make sense for empty queues! They are functions with non-trivial pre-conditions, which are undefined when the pre-condition is unsatisfied. Their Haskell definitions contain inexhaustive pattern matching, which is translated into Core **case** expressions with calls to the `error` function in one branch. The `error` function is comparable to non-termination, in the sense that it does not produce a result, and so cannot be translated directly into Agda. Functions with non-trivial preconditions are common, and thus pose a much more immediate translation problem than do infinite loops.

4. A Monadic Translation of Haskell to Agda

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 `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[2]:

$$
\begin{array}{r}
m : \text{Set} \to \text{Set} \\
\mathit{return} : (\alpha : \text{Set}) \mapsto \alpha \to m\,\alpha \\
(\gg\!=) : (\alpha : \text{Set}) \mapsto (\beta : \text{Set}) \mapsto m\,\alpha \to (\alpha \to m\,\beta) \to m\,\beta
\end{array}
$$

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

$$
\begin{aligned}
\alpha^\dagger &= m\,\alpha \\
(\tau_1 \to \tau_2)^\dagger &= m(\tau_1^\dagger \to \tau_2^\dagger) \\
x^\dagger &= x \\
(\lambda x.e)^\dagger &= \mathit{return}(\lambda x.e^\dagger) \\
(e_0 e_1)^\dagger &= e_0^\dagger \gg\!= \lambda f.f e_1^\dagger
\end{aligned}
$$

With this translation, function arguments are translated into monadic computations, which can thus be `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 λ-abstractions, so that

$$
\begin{aligned}
(\forall \alpha.\tau)^\dagger &= m((\alpha : \text{Set}) \to \tau^\dagger) \\
(\Lambda \alpha.e)^\dagger &= \mathit{return}(\lambda \alpha.e^\dagger) \\
(e\ @\ \tau)^\dagger &= e^\dagger \gg\!= \lambda f.f \tau^\dagger
\end{aligned}
$$

This was the approach taken by Barthe, Hatcliff, 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 *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 $\forall \alpha.\alpha \to \alpha$, 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 Set $\to$ Set. But in the BHT translation, m is applied to the type $(\alpha : \text{Set}) \to \tau^\dagger$—which cannot be in Set, because it involves Set itself! Redefining m with type Type $\to$ Type (where Type is the next universe beyond Set) does not help, because the monad m will also appear in the types which we instantiate α to. So if m were of type Type $\to$ Type, then we would need to abstract over Type instead of 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

[2] In this section we write Agda with a "mathematical" notation.

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 $(\Lambda\alpha.e)(\forall\beta.\tau)$, 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 polymorphism 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 translated 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 elements of 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 Set as arguments, we will need a different translation which does not involve the monad. We shall distinguish between the types of Core functions which are translated monadically, and those which are not, by writing the latter in the form $\sigma_1 \mapsto \sigma_2$. 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 $\mapsto$, introducing the latter for functions taking either types or class dictionaries (which may have polymorphic types) as parameters. Thus, the types in our annotated dialect of Core are generated from the following grammar:

$$\begin{array}{ll} \tau ::= \alpha \mid D \mid \tau\tau \mid \tau \rightarrow \tau & \text{monotypes} \\ \sigma ::= \tau \mid \forall\alpha : \kappa.\,\sigma \mid \sigma \mapsto \sigma & \text{polytypes} \end{array}$$

Since the translation of $\mapsto$ functions is non-monadic, we can also think of them as "unlifted" functions, for which we do not distinguish 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 distinguish 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:

$$\begin{aligned} *^\dagger &= \text{Set} \\ (\kappa_1 \rightarrow \kappa_2)^\dagger &= \kappa_1^\dagger \rightarrow \kappa_2^\dagger \end{aligned}$$

Note that, for any kind κ, we have $\vdash \kappa^\dagger$: Type—its translation is an Agda Type. Now we can summarize the translation of Core types:

$$\begin{aligned} (\forall\alpha : \kappa.\,\sigma)^\dagger &= (\alpha : \kappa^\dagger) \rightarrow \sigma^\dagger \\ (\sigma_1 \mapsto \sigma_2)^\dagger &= \sigma_1^\dagger \rightarrow \sigma_2^\dagger \\ \tau^\dagger &= m\,\tau^* \end{aligned}$$

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

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

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

The translation of Core λ-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 λ-abstractions and applications depends on whether the function is of mono- or polytype—functions with $\mapsto$ types are translated into functions, while functions of $\rightarrow$ types are translated into monadic computations.

These rules also depend on a translation of environments:

$$\begin{aligned} \{\}^\dagger &= \{\} \\ (\Gamma, \alpha : \kappa)^\dagger &= \Gamma^\dagger, \alpha : \kappa^\dagger \\ (\Gamma, x : \sigma)^\dagger &= \Gamma^\dagger, x : \sigma^\dagger \end{aligned}$$

Note that the monad m is only applied to elements of Set in the translated code!

Data-type definitions pose a special problem. They must be translated into Agda data-type definitions, but constructors in Haskell (and thus in Core) are just functions with a type of the form $\forall\vec{\alpha} : \vec{\kappa}.\tau_1 \rightarrow \cdots \rightarrow \tau_k \rightarrow D\ \vec{\alpha}$. The translation of such a type is of the form $(\vec{\alpha} : \vec{\kappa}^\dagger) \rightarrow m\ (\tau_1^\dagger \rightarrow \ldots m\ (\tau_k^\dagger \rightarrow m\ (D\ \vec{\alpha}))$, and of course, no Agda constructor can have such a type. Therefore, Haskell constructors cannot be translated directly into Agda constructors. Instead, we will introduce Agda constructors with types of the form $(\vec{\alpha} : \vec{\kappa}^\dagger) \rightarrow \tau_1^\dagger \rightarrow \ldots \tau_k^\dagger \rightarrow D\ \vec{\alpha}$, 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:

$$\Gamma \vdash d : \Delta$$

$$\Gamma; \vec{\alpha} : \vec{\kappa}; D : \vec{\kappa} \rightarrow * \vdash cd : \Delta$$

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.

Luckily, Core permits only full applications of constructors[5] to all of their arguments—partial applications in Haskell are η-converted to λ-expressions during translation to Core. Thus we need only translate full applications to Agda:

$$\left[\Gamma \vdash C @ \vec{\tau}\,\vec{e} : D\vec{\tau}\right]^\dagger = \Gamma^\dagger \vdash \mathit{return}\ (C\,\vec{e}^\dagger) : m(D\vec{\tau}^\dagger)$$

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

[3] This kind of term appears only when the programmer uses an existential type, or a datatype with a polymorphic component. Our method cannot translate such programs into Agda, but fortunately they are fairly rare.

[4] For example, the dictionary associated with the ubiquitous `Monad` class has a polymorphic field, the implementation of bind.

[5] Contrary to Haskell, Core constructors cannot have strict fields—if a constructor is strict in an argument in Haskell, that argument is evaluated explicitly before the constructor is applied in Core.

$$\left[\frac{}{\Gamma, x:\sigma \vdash x:\sigma}\right]^{\dagger} = \frac{}{\Gamma^{\dagger}, x:\sigma^{\dagger} \vdash x:\sigma^{\dagger}}$$

$$\left[\frac{\Gamma, x:\tau_1 \vdash e:\tau_2}{\Gamma \vdash \lambda(x:\tau_1).\, e:\tau_1 \to \tau_2}\right]^{\dagger} =$$

$$\frac{\Gamma^{\dagger}, x:\tau_1^{\dagger} \vdash e^{\dagger}:\tau_2^{\dagger}}{\Gamma^{\dagger} \vdash \mathit{return}(\lambda(x:\tau_1^{\dagger}).e^{\dagger}) : m(\tau_1^{\dagger} \to \tau_2^{\dagger})}$$

$$\left[\frac{\Gamma \vdash e_0:\tau_1 \to \tau_2 \quad \Gamma \vdash e_1:\tau_1}{\Gamma \vdash e_0 e_1 : \tau_2}\right]^{\dagger} =$$

$$\frac{\Gamma^{\dagger} \vdash e_0^{\dagger} : (\tau_1 \to \tau_2)^{\dagger} \quad \Gamma^{\dagger} \vdash e_1^{\dagger} : \tau_1^{\dagger}}{\Gamma^{\dagger} \vdash e_0^{\dagger} \gg\!= \lambda(f:\tau_1^{\dagger} \to \tau_2^{\dagger}).f e_1^{\dagger} : \tau_2^{\dagger}}$$

$$\left[\frac{\Gamma, x:\sigma_1 \vdash e:\sigma_2}{\Gamma \vdash \lambda(x:\sigma_1).\, e:\sigma_1 \mapsto \sigma_2}\right]^{\dagger} =$$

$$\frac{\Gamma^{\dagger}, x:\sigma_1^{\dagger} \vdash e^{\dagger}:\sigma_2^{\dagger}}{\Gamma^{\dagger} \vdash \lambda(x:\sigma_1^{\dagger}).e^{\dagger} : \sigma_1^{\dagger} \to \sigma_2^{\dagger}}$$

$$\left[\frac{\Gamma \vdash e_0:\sigma_1 \mapsto \sigma_2 \quad \Gamma \vdash e_1:\sigma_1}{\Gamma \vdash e_0 e_1 : \sigma_2}\right]^{\dagger} =$$

$$\frac{\Gamma^{\dagger} \vdash e_0^{\dagger} : \sigma_1^{\dagger} \to \sigma_2^{\dagger} \quad \Gamma \vdash e_1^{\dagger} : \sigma_1^{\dagger}}{\Gamma^{\dagger} \vdash e_0^{\dagger} e_1^{\dagger} : \sigma_2^{\dagger}}$$

$$\left[\frac{\Gamma, \alpha:\kappa \vdash e:\sigma}{\Gamma \vdash \Lambda(\alpha:\kappa).\, e:\forall\alpha:\kappa.\sigma}\right]^{\dagger} =$$

$$\frac{\Gamma^{\dagger}, \alpha:\kappa^{\dagger} \vdash e^{\dagger}:\sigma^{\dagger}}{\Gamma^{\dagger} \vdash \lambda(\alpha:\kappa^{\dagger}).e^{\dagger} : (\alpha:\kappa^{\dagger}) \to \sigma^{\dagger}}$$

$$\left[\frac{\Gamma \vdash e:\forall\alpha:*.\sigma \quad \Gamma \vdash \tau:*}{\Gamma \vdash e \mathbin{@} \tau : \sigma[^{\tau}/_{\alpha}]}\right]^{\dagger} =$$

$$\frac{\Gamma^{\dagger} \vdash e^{\dagger} : (\alpha:\mathrm{Set}) \to \sigma^{\dagger} \quad \Gamma^{\dagger} \vdash \tau^{*} : \mathrm{Set}}{\Gamma^{\dagger} \vdash e^{\dagger}\tau^{*} : \sigma^{\dagger}[^{\tau^{*}}/_{\alpha}]}$$

Figure 4. Translation of abstractions and applications.

The translation of **case** expressions appears in Figure 6. Clauses alt are typed using a four-ary judgment $\Gamma \vdash alt : \tau_1 \Rightarrow \tau_2$, where τ_1 is the type of the pattern and τ_2 the type of the branch. We use monadic $\gg\!=$ to evaluate the expression the **case** inspects, so the actual Agda case analysis of the value appears in the second argument of $\gg\!=$. 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 $\gg\!=$ 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

$$f : a \to b$$
$$f = \lambda(x:a).e$$

$$\left[\frac{\{\Gamma; \vec{\alpha}:\vec{\kappa}; D:\vec{\kappa} \to * \vdash cd_i : \Delta_i\}_{i=1}^{n}}{\begin{array}{l}\Gamma \vdash \ \ \mathbf{data}\ D(\vec{\alpha}:\vec{\kappa}) = cd_1 \mid \ldots \mid cd_n \\ \qquad : \{D:\vec{\kappa} \to *\} \cup \Delta_1 \cup \cdots \cup \Delta_n\end{array}}\right]^{\dagger} =$$

$$\frac{\{\Gamma^{\dagger}; \vec{\alpha}:\vec{\kappa}^{\dagger}; D:\vec{\kappa}^{\dagger} \to \mathrm{Set} \vdash cd_i^{\dagger} : \Delta_i^{\dagger}\}_{i=1}^{n}}{\begin{array}{l}\Gamma^{\dagger} \vdash \ \ \mathbf{data}\ D(\vec{\alpha}:\vec{\kappa}^{\dagger}) = cd_1^{\dagger} \mid \ldots \mid cd_n^{\dagger} \\ \qquad : \{D:\vec{\kappa}^{\dagger} \to \mathrm{Set}\} \cup \Delta_1^{\dagger} \cup \cdots \cup \Delta_n^{\dagger}\end{array}}$$

$$\left[\frac{\{\Gamma, \vec{\alpha}:\vec{\kappa}, D:\vec{\kappa} \to * \vdash \tau_i : *\}_{i=1}^{n}}{\begin{array}{l}\Gamma; \vec{\alpha}:\vec{\kappa}; D:\vec{\kappa} \to * \vdash \\ \quad C\tau_1 \ldots \tau_n : \{C : \forall\vec{\alpha}:\vec{\kappa}.\tau_1 \to \cdots \to \tau_n \to D\vec{\alpha}\}\end{array}}\right]^{\dagger} =$$

$$\frac{\{\Gamma^{\dagger}, \vec{\alpha}:\vec{\kappa}^{\dagger}, D:\vec{\kappa}^{\dagger} \to \mathrm{Set} \vdash \tau_i^{\dagger} : \mathrm{Set}\}_{i=1}^{n}}{\begin{array}{l}\Gamma^{\dagger}; \vec{\alpha}:\vec{\kappa}^{\dagger}; D:\vec{\kappa}^{\dagger} \to \mathrm{Set} \vdash \\ \quad C(v_1:\tau_1^{\dagger}) \ldots (v_n:\tau_n^{\dagger}) : \\ \qquad \{C : (\vec{\alpha}:\vec{\kappa}^{\dagger}) \to \tau_1^{\dagger} \to \cdots \to \tau_n^{\dagger} \to D\vec{\alpha}\}\end{array}}$$

where $v_1, \ldots, v_n$ are fresh variables.

Figure 5. Translation of data declarations.

$$\left[\frac{\Gamma \vdash e:\tau_0 \quad \Gamma \vdash alts:\tau_0 \Rightarrow \tau}{\Gamma \vdash \mathbf{case}\ e\ \mathbf{of}\ alts : \tau}\right]^{\dagger} =$$

$$\frac{\Gamma^{\dagger} \vdash e^{\dagger}:\tau_0^{\dagger} \quad \Gamma^{\dagger} \vdash alts^{\dagger} : \tau_0^{*} \Rightarrow \tau^{\dagger}}{\Gamma^{\dagger} \vdash \mathbf{let}\ f(x:\tau_0^{*}) : \tau^{\dagger} = \mathbf{case}\ x\ \mathbf{of}\ alts^{\dagger}\ \mathbf{in}\ e \gg\!= f : \tau^{\dagger}}$$

$$\left[\frac{\Gamma, \vec{x}:\vec{\tau} \vdash C@\vec{\tau}'\vec{x} : D\vec{\tau}' \quad \Gamma, \vec{x}:\vec{\tau} \vdash e:\tau}{\Gamma \vdash C(\vec{x}:\vec{\tau}) \to e : D\vec{\tau}' \Rightarrow \tau}\right]^{\dagger} =$$

$$\frac{\Gamma^{\dagger}, \vec{x}:\vec{\tau}^{\dagger} \vdash C\vec{x} : D\vec{\tau}'^{\dagger} \quad \Gamma^{\dagger}, \vec{x}:\vec{\tau}^{\dagger} \vdash e^{\dagger}:\tau^{\dagger}}{\Gamma^{\dagger} \vdash C\vec{x} \to e^{\dagger} : D\vec{\tau}'^{\dagger} \Rightarrow \tau^{\dagger}}$$

Figure 6. Translation of case expressions.

The translation of this definition is

$$f : m(a^{\dagger} \to b^{\dagger})$$
$$f = \mathit{return}\ \lambda(x:a^{\dagger}).e^{\dagger}$$

Note that f itself is assigned a monadic type, and that applications $f\ e$ are translated to $f \gg\!= \lambda\overline{f}.\overline{f}\ e^{\dagger}$. 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 $\perp$, and so assigning it a monadic type and invoking it via $\gg\!=$ is just overkill. Our translator therefore omits the application of the monad to the λ-expression, generating the optimised translation

$$f : a^{\dagger} \to b^{\dagger}$$
$$f = \lambda(x:a^{\dagger}).e^{\dagger}$$

instead. We restrict the applications of f to be full applications (η-converting if need be), which can then be translated simply via $(f\ e)^{\dagger} = f\ e^{\dagger}$. 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 natural 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:

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

(++) :: (a :: Set) |-> m (List a)-> m (List a)->
                       m (List a)
  = \a |->
     let app1 :: m (List a)-> m (List a)->
                 m (List a)
           = \mxs mys ->
              let app2 (xs :: List a):: m (List a)
                    = case xs of
                        (Nil)-> mys
                        (Cons x xs')->
                              return (Cons x
                                     (app1 xs' mys))
             in mxs >>= app2
    in app1
```

(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:

```
m :: Set -> Set = \a -> a
return :: (a::Set) |-> a -> m a = \a |-> \x -> x
(>>=) :: (a,b::Set) |-> m a -> (a -> m b) -> m b
      = \a b |-> \ma -> \mf-> mf ma
```

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

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

```
reflEqList :: (a::Set) |-> (eq:: a -> a -> Set) ->
              (refl:: (x::a) -> eq x x) ->
              (xs::List a) -> Eq eq xs xs
```

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

```
EqS   :: s -> s -> Set
reflS :: (x :: s) -> EqS x x
```

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

```
app_assoc :: (xs,ys,zs::List s) ->
             Eq EqS (xs ++ (ys ++ zs))
                    ((xs ++ ys) ++ zs)
  = \xs ys zs ->
     case xs of
          (Nil)-> reflEqList EqS reflS (ys ++ zs)
          (Cons x xs')-> and (reflS x)
                             (app_assoc xs' ys zs)
```

When the first argument of append is empty, the definition of append simple returns the second argument. Hence, when `xs` is empty the property amounts to proving that `ys ++ zs` is equal to itself, which is true by the reflexivity of the equality on lists. On the other hand, if `xs` is 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' ++ ys) ++ 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:

```
m :: Set -> Set = \a -> Maybe a
return :: (a::Set) |-> a -> m a
   = \a |-> \x -> Just x
(>>=) :: (a,b::Set) |-> m a -> (a -> m b) -> m b
   = \a b |-> \ma -> \mf ->
      case ma of (Nothing)-> Nothing
                 (Just x )-> mf x
```

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).

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

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

reflEqM :: (a::Set) |-> (eq::a -> a -> Set) ->
           (refl :: (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 ++ (mys ++ mzs))
                          ((mxs ++ mys) ++ mzs)
 = \mxs mys mzs ->
    case mxs of
      (Nothing)-> tt
      (Just xs)->
         case xs of
           (Nil)-> reflEqM (Eq EqS)
                           (reflEqLst EqS reflS)
                           (mys ++ mzs)
           (Cons ma mas)->
                 and (reflEqM EqS reflS 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 `mxs` we obtain the cases `Nothing` and `Just xs`. When `mxs` 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, `mxs` 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 |-> \mx -> \mqs ->
    mqs ++ (return (Cons mx (return Nil)))
```

One 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 `addI` exemplifies these two points. Here, GHC has inlined the application to the function `flipQ` and replaced the type of queues by the type of pairs of lists.

```
addI::(a::Set) |-> m a ->
                  m (Pair (List a) (List a)) ->
                  m (Pair (List a) (List a))
 = \a |-> \ma -> \mp ->
   let addI1 (p :: Pair (List a) (List a))
             :: m (Pair (List a) (List a))
         = case p of
             (P f b)->
               let addI2 (xs :: List a)
                    :: m (Pair (List a) (List a))
                    = case xs of
                        (Nil)->
                          return
                            (P (reverse
                                 (return
                                   (Cons ma b)))
                               (return Nil))
                        (Cons mx mxs)->
                          return (P (return xs)
                                    (return (Cons ma b)))
                in f >>= addI2
   in mp >>= addI1
```

However, the translations of the functions returning the first element of a non-empty queue (`front` and `frontI`) or the remaining part of a non-empty queue after removing its first element (`remove` and `removeI`) 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 `flipQ`.

```
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
QueueI (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 :: QueueI 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->
    let mf = mfst mp; mb = msnd mp
    in And3 (TM (totalLst mf))
            (TM (totalLst mb))
          (Or (TM (null mb)) (TM (not (null mf))))
```

Here, `mfst` and `msnd` select the first and second element, respectively, of a Maybe pair, `And3` is defined similarly to `And` but it performs 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 s) -> (mq :: m (QueueI s)) ->
             EqM (Eq EqS) (retrieve (addI ma mq))
                          (add ma (retrieve 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
                            (Just (Cons ma mb)))
                  (Cons mx mxs)->
                    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, `app_nil mxs` is a proof that `mxs ++ (return Nil)` is equal to `mxs`, for `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))->
    \(inv::invariant mp)->
    \(ne::TM (not (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 xs)->
                  case xs of
                   (Nil)-> case ne of { }
                   (Cons mx mxs)->
                     case mxs of
                      (Nothing)-> case tf of { }
                      (Just xs')->
                        case xs' of
                         (Nil)->
                          and3 (tot2rev_tot mb tb)
                               tf  (inl tt)
                         (Cons mx' mxs')->
                               and3 tf tb (inr tt)
```

Since the invariant of `mp` is true, `mp` cannot be `Nothing`, and neither can it contain a sub-part that is `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 `Just (P (Just (Cons mx (Just xs')))) mb`, for the cases where `xs'` is empty and it is not empty. Both cases are easy. Here, `tot2rev_tot` 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 `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 `NOINLINE`.[6]

6. Related Work

The monadic translation of Barthe, Hatcliff, and Thiemann [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, binary sums, and induction. Using his approach directly would insert too many applications of the monad for our purposes, therefore we have our own translation of data types which is better suited for practical applications.

Verification for pure functional languages. De Mol, Van Eekelen, and Plasmeijer [7], present SPARKLE, a theorem prover optimized for the functional programming language CLEAN. SPARKLE operates on CORE-CLEAN, a fragment of CLEAN comparable to Haskell-Core. The proof about a CLEAN program is performed interactively on the translated program; it is claimed that the translation does not obfuscate the code (except for list comprehension). This did not become entirely clear to us, since the running example in the paper does not use advanced non-core features, such as type classes, which are translated via dictionaries. In comparison to CLEAN proofs, which are tactic scripts, Agda proofs are λ-terms, which are understandable independently from the proof tool. While a special purpose theorem prover such as SPARKLE might provide some comfort for the user, we rely on an existing prover with a well-understood meta theory whose soundness is backed by a long theoretical tradition.

The *Programatica* project aims at certifying properties of Haskell programs, where certificates are not restricted to formal proofs, but could also be test certificates or references to literature where properties of an algorithm have been proven on a more abstract level. Harrison and Kieburtz [10] describe P-logic, a verification logic for Haskell based on the modal μ-calculus, in which recursive invariants of data structures can be concisely expressed.

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

P-logic is especially tuned to the strict and lazy aspects of Haskell semantics and has a definedness modality '$'. P-logical properties are mixed with Haskell source and separated out by the syntactical tool *Programatica front end*. Hence, P-logic has in principle to deal with *all* of Haskell syntax, which we avoid by working with Haskell core. It seems that advanced features like type classes, which are translated away by GHC in our case, are not yet covered by P-logic.

Hallgren [9] has implemented a translation of Haskell into Alfa, a graphical front-end to the Agda proof language. He translates type classes via dictionaries, but does not address the problems of partiality and non-termination. Haskell code is more or less literally mapped to Agda code which jeopardizes the soundness of the type theory in the presence of partiality. By assuming that all Agda types are inhabited via a postulate $\mathit{undefined} : (A : \mathrm{Set}) \to A$ he can translate partial functions, but if this fact is used for propositions, anything is true. In manual proofs, one can avoid using $\mathit{undefined}$, but automated proof search using the *Agda synthesizer Agsy* could not be performed without changes to the system.

Uses of interactive theorem provers for verification. Filliâtre [8] uses the Calculus of Inductive Constructions to verify Hoare-logical properties of imperative programs.

Kreitz [12] has embedded a significant subset of Ocaml, including references and exceptions, into Nuprl in the style of denotational semantics. He uses the syntax extension mechanism of Nuprl to display the translated code in actual Ocaml syntax, and obtains derived typing and computation rules for Ocaml programs. As an application [13], he provides a framework for performing provably correct optimizations of network protocol stacks in the communication toolkit ENSEMBLE, which is implemented in Ocaml.

Longley and Pollack [15] use Isabelle/HOL as a framework to represent the functional core of Standard ML. By Isabelle's axiomatic type classes they define a class of SML types which are all inhabited by the bottom element. Two predicates characterize defined (non-bottom) and undefined SML expressions. This way they can handle partiality and infinite data structures. A serious problem is Hilbert's epsilon operator, which is available in Isabelle's classical logic for all types, even the SML types, and enables one to define non-continuous functions (e.g., the function which swaps the unit and bottom element).

7. Conclusions

Proving programs correct using an existing prover requires us to build a model of the program within the prover. We have shown that, surprisingly, the intermediate code generated by GHC can serve as a suitable base for proofs in a type theoretic theorem prover. We have developed a new, and natural, monadic translation that lets us reason about partial values, or ignore them, as we choose. Proofs about total values are not complicated at all by the presence of the monad, and proofs about partial values just include the extra $\perp$-cases one would expect—this because the monadic translation is carefully designed to be reduced away by the prover's type-checker. Although Agda's predicativity limits the programs we can translate, in practice almost all Haskell programs are translatable. Proofs of Haskell programs can be performed just with reference to the Haskell source code, not its translation, and are no more complex than proofs of an Agda model constructed by hand would be.

Proofs on a larger scale will require more automation. Current work in this direction includes Agsy, a plug-in for Agda which searches for type-theory proofs, and a first-order logic plug-in which delegates sub-goals to an external prover.

An important future goal is to *combine* reasoning with and without partiality. At present, a partial function can only be interpreted 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 *CoVer* project: Thierry and Catarina 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 Classen, Peter Dybjer, and Mary Sheeran for discussions on proofs about partial programs. We are indepted 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

[1] G. Barthe, J. Hatcliff, and P. Thiemann. Monadic type systems: Pure type systems for impure settings. In A. Gordon, A. Pitts, and C. Talcott, editors, *HOOTS II, Second Workshop on Higher-Order Operational Techniques in Semantics*, volume 10 of *Electronic Notes in Theoretical Computer Science*, pages 54–120, 1997.

[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Texts in Theoretical Computer Science. Springer-Verlag, 2004,.

[3] V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 2005. To appear.

[4] K. Claessen and J. Hughes. Quickcheck: A lightweight tool for random testing of haskell programs, 2000.

[5] C. Coquand and T. Coquand. Structured type theory. In *Proc. Workshop on Logical Frameworks and Meta-languages*, 1999.

[6] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[7] M. de Mol, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Theorem proving for functional programmers. In T. Arts and M. Mohnen, editors, *Implementation of Functional Languages, 13th International Workshop, IFL 2001, Stockholm, Sweden, September 24-26, 2001*, Lecture Notes in Computer Science, pages 55–71. Springer, 2002.

[8] J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.

[9] T. Hallgren, J. Hook, M. P. Jones, and R. B. Kieburtz. An overview of the programatica toolset. Presented at the High Confidence Software and Systems Conference, HCSS04, 2004.

[10] W. L. Harrison and R. B. Kieburtz. A logic of demand in Haskell. *Journal of Functional Programming*, 2005. Under consideration of publication.

[11] W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, London, 1980.

[12] C. Kreitz. Formal reasoning about communication systems I: Embedding ML into type theory. Technical report, Cornell University, 1997.

[13] C. Kreitz. Building reliable, high-performance networks with the Nuprl proof development system. *Journal of Functional Programming*, 14(1):21–68, 2004.

[14] D. Leivant. Finitely stratified polymorphism. *Information and Computation*, 93(1):93–113, July 1991.

[15] J. Longley and R. Pollack. Reasoning about cbv functional programs in isabelle/hol. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004, Proceedings*, volume 3223 of *Lecture Notes in Computer Science*, pages 201–216. Springer, 2004.

[16] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Napoli, 1984.

[17] F. Pfenning and C. Schürmann. System description: Twelf — A meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, 1999. Springer-Verlag LNAI 1632.

[18] T. Uustalu. Monad translating inductive and coinductive types. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*, pages 299–315. Springer, 2003.

[19] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, Jan. 1989.