Proving Non-Deterministic Computations in Agda*

Sergio Antoy
Computer Science Dept.
Portland State University
Oregon, U.S.A.
antoys@pdx.edu

Michael Hanus
Institut für Informatik
CAU Kiel
Germany
mh@informatik.uni-kiel.de

Steven Libby
Computer Science Dept.
Portland State University
Oregon, U.S.A.
slibby@pdx.edu

We investigate proving properties of Curry programs using Agda. First, we address the functional correctness of Curry functions that, apart from some syntactic and semantic differences, are in the intersection of the two languages. Second, we use Agda to model non-deterministic functions with two distinct and competitive approaches incorporating the non-determinism. The first approach eliminates non-determinism by considering the set of all non-deterministic values produced by an application. The second approach encodes every non-deterministic choice that the application could perform. We consider our initial experiment a success. Although proving properties of programs is a notoriously difficult task, the functional logic paradigm does not seem to add any significant layer of difficulty or complexity to the task.

1 Introduction

The growing interest in software correctness has led to a variety of approaches for proving, or increasing one's confidence, that a program computes what is intended. Examples of this include *formal verification* [11], *design by contract* [22], *program analysis* [23] and *model checking* [21]. We are interested in the *functional correctness* of a program, i.e., the property that each input to the program produces the intended output. Agda [24], a language and system based on *dependent types*, is designed for this purpose.

Agda programs define types and functions. Types are seen as statements, and functions as constructive proofs of those statements according to the Curry-Howard isomorphism [15]. Agda functions resemble those of Haskell [26]. Agda types extend Haskell's type system since definitions can be parameterized by (depend on) values of other types. We are interested in proving properties of Curry programs [18]. Curry programs define types and functions that resemble those of Haskell as well, but with a key difference: A "function" in Curry can be non-deterministic. Loosely speaking, this means that for the same input, the function can return one of a set of values. These *non-deterministic functions* are not functions in the mathematical sense, but using them as functions is convenient. In particular, functional application and functional nesting provide optimally lazy non-deterministic computations [1].

For example, the following Curry code defines a function `perm`, that takes a list of elements and returns any permutation of the list non-deterministically.

```
perm []     = []
perm (x:xs) = ndinsert x (perm xs)
ndinsert x []     = x:[]
ndinsert x (y:ys) = x:y:ys ? y:ndinsert x ys
```

The question mark in the last rule denotes the *choice* operation defined by the rules:

*This material is based upon work partially supported by the National Science Foundation under Grant No. CCF-1317249 and the German Federal Ministry of Education and Research (BMBF) under Grant No. 01IH15006B.

```
x ? y = x
x ? y = y
```

The semantics of this definition differs from Agda's and Haskell's. In the full spirit of declarative programming, there is no textual order among these rules. Thus, a value of $t \,?\, u$ is any value of t or any value of u. The *choice* is substantially equal to McCarthy's *amb* operator [20].

Our goal is using Agda to prove the correctness of Curry programs. This must take into account where Curry differs from Adga, e.g., non-determinism, the order of evaluation, and non-termination. Non-determinism may cause some computations to fail, and non-termination is essential for Curry's lazy evaluation. While all these differences are relevant, the crucial one to handle is non-determinism.

First, we discuss a class of programs that substantially is the same in Agda and Curry, hence proving the functional correctness of an Agda program in this class proves it for Curry as well. Then, we discuss two techniques to cast non-deterministic Curry programs into programs in the previous class, hence extending proofs to non-deterministic programs. The proofs and libraries discussed in this paper are available at `github.com/mihanus/curry-agda`.

2 Curry

Before discussing our techniques for proving properties of Curry programs, we briefly review some aspects of functional logic programming, and Curry, that are necessary to understand the content of this paper. More details can be found in recent surveys on functional logic programming [6, 17].

Curry [18] is a declarative, multi-paradigm language, intended to combine the most important features of functional and logic programming. The syntax of Curry is close to Haskell [26], only it is extended by allowing *free* (*logic*) *variables* in conditions and in right-hand sides of rules. Moreover, the patterns of a defining rule can be non-linear, i.e., they might contain multiple occurrences of a variable, which is an abbreviation for equalities between these occurrences.

The following simple program shows the functional and logic features of Curry. It defines an operation "++" to concatenate two lists, which is identical to the Haskell encoding. The second operation, `ndins`, shows an alternative definition of the non-deterministic list insertion shown in the introduction.[1]

```
(++) :: [a] → [a] → [a]          ndins :: a → [a] → [a]
[]     ++ ys = ys                ndins x xs | xs == ys++zs
(x:xs) ++ ys = x : (xs ++ ys)               = ys++[x]++zs   where ys,zs free
```

Function calls can contain free variables. They are evaluated lazily where free variables, as demanded arguments, are non-deterministically instantiated. Since it is also known that free variables can be replaced by non-deterministic functions (value generators) [5], we mainly consider non-deterministic functions instead of free variables in this paper. Another major difference from Haskell is that the rules defining an operation are not applied in their textual order. Any rule which is applicable to some expression is applied non-deterministically to evaluate this expression. Thus, if Curry evaluates a call to the choice operation "?" defined in the introduction, both rules are applicable. Hence, the expression "`0 ? 1`" evaluates to `0` and `1` with the value non-deterministically chosen.

Although non-deterministic functions are a useful programming feature that leads to new design patterns [7], they also cause semantic subtleties that need to be clarified. For instance, consider the following function to double the value of an integer number:

```
double x = x + x
```

[1]Note that Curry requires the explicit declaration of free variables, as `ys` and `zs` in the rule of `ndins`, to ensure checkable redundancy.

One would expect that the result of this operation is always an even number. Actually, we will show in Sect. 6 how to prove this property with Agda. However, it is not obvious that this property holds with respect to non-deterministic computations. For instance, one might consider the following derivation:

```
double (0?1)  → (0?1) + (0?1)  → 0 + (0?1)  → 0 + 1  → 1
```

In this derivation, we use two different values for a single argument. Although one might consider such derivations as acceptable, which is known as a *run-time choice* semantics, they come with a potential problem. The result of such computations depends on the evaluation strategy—an aspect that should be avoided in a declarative language. For instance, `1` cannot be obtained as a result of `double (0?1)` with respect to an eager (innermost) evaluation. Therefore, Curry and similar functional logic languages adapt the *call-time choice* semantics [19] to obtain a strategy-independent combination of functional and non-deterministic programming. Intuitively, an argument passed to a function always has a unique value. This does not mean that one cannot use non-deterministic functions as arguments, but it implies that a call like `double (0?1)` is interpreted like two calls, `double 0` and `double 1`. The details of this semantics, known as the CRWL rewriting logic, can be found in [16].

3 Agda

Agda is a dependently typed functional language, [9, 25] and is based on intuitionistic type theory. Similar to Curry, Agda datatypes are introduced by a data declaration, which gives the name and type of the datatype as well as the constructors and values in that type. Agda's types are much more general than Curry's, because type definitions may depend on values, e.g., the type of lists of a given length. Agda functions are also required to produce a result for every application. Thus, a function definition must cover a complete set of patterns and an application must terminate for every argument. The parameter passing mode of a function application is by-value, i.e., the arguments are normalized before the application. Very loosely speaking, Adga has a richer set of types, whereas Curry has a richer set of functions.

An Agda program typically defines some types, and functions over these types. A later example defines the functions "+", which takes two natural numbers (in unary representation) and returns their sum, and `even`, which takes a natural number and returns a Boolean value telling whether the argument is an even number. A third function, whose identifier is `even-x+x`, takes a natural x, and has the return type `even` $(x$ `+` $x)$ $\equiv$ `tt`, where `tt` is the Boolean value *true*, "$\equiv$" is the propositional equality, and "+" and `even` were previously defined. According to the Curry-Howard isomorphism [15], the existence of a function of this type, witnessed by its definition, proves the statement captured by the return type: for every natural x, $x+x$ is even.

Our goal is to prove statements about Curry functions. Thus, we will define types that capture the statements we wish to prove, and encode, when we can, definitions of functions of these types. These will be the proofs of the statements. An obvious problem is that Curry allows us to encode functions that cannot be encoded in Agda, and even those that are syntactically similar in the two languages have significant semantic differences. Therefore, we will restrict the functions we consider, and discuss the validity of the statements that we prove despite the semantic differences.

4 Non-determinism

In this section we discuss two approaches to model non-deterministic computations in Agda. Consider the function `perm` defined in the introduction. The Curry evaluation of `perm [1,2,3]` may return

[2,1,3], or any of the six permutations of the argument. Suppose that we wish to prove that for any list L, perm L has the same length as L. A crucial task is modeling perm in Agda, which does not allow non-determinism. To understand our approaches, let us first develop a hypothetical computation of perm [1,2,3] in Curry. We take some liberties that will rectify shortly:

```
perm [1,2,3] → ndinsert 1 (perm [2,3])              (1)
             → ndinsert 1 [2,3]                     (2)
             → [1,2,3] ? 2 : ndinsert 1 [3]         (3)
             → 2 : ndinsert 1 [3]                   (4)
             → [2,1,3]                              (5)
```

In line (1) we eagerly evaluate perm [2,3] and arbitrarily assume that it returns [2,3]. In line (3) we arbitrarily choose the right argument of "?". In line (4) we eagerly evaluate ndinsert 1 [3], and arbitrarily assume that it returns [1,3].

Since we are going to code perm in Agda, we must adopt eager evaluation. In Curry the order of evaluation does not affect the result of a computation, as long as the result is obtained. Thus, for strongly terminating programs, the functional correctness of a program is independent of eager vs. lazy evaluation. The real problem is to model the somewhat arbitrary choices made by non-deterministic functions.

4.1 Set of values

Our first approach encodes a Curry non-deterministic function f into an Agda function f', which must be deterministic. For any given argument, f' returns the set of all the values that f could possibly return. To this aim, we declare a set-like structure as follows:

```
data ND (A : Set) : Set where
  Val  : A → ND A
  _??_ : ND A → ND A → ND A
```

The constructor Val abstracts a deterministic value in the set. The constructor "??" makes a set out of two sets, and captures a non-deterministic choice.

For example, type ND is used to code an Agda function, perm, that computes the set of all the permutations of its argument.

```
ndinsert : {A : Set} → A → 𝕃 A → ND (𝕃 A)
ndinsert x []        = Val (x :: [])
ndinsert x (y :: ys) = Val (x :: y :: ys)
                     ?? (_::_ y) $* (ndinsert x ys)

perm : {A : Set} → 𝕃 A → ND (𝕃 A)
perm []        = Val []
perm (x :: xs) = ndinsert x *$* (perm xs)
```

The functions "$*" and "*$*" are defined in a library that we have developed for our purpose (see Sect. 5). Function "$*" is a standard *map* function over the ND structure, i.e., it applies the first argument, a deterministic function, to every value of the second. Function "*$*" is also a *map*, but its first argument is a non-deterministic function.

With this infrastructure, the statement that any permutation of a list L has the same length as L is formalized as follows:

```
perm-length : {A : Set} → (xs : 𝕃 A) →
  (perm xs) satisfy (λ ys → length ys =ℕ length xs) ≡ tt
```

In the above statement, `satisfy` is another library function, which is infix, with the meaning intended by its name. The symbols "$=\mathbb{N}$" and "$\equiv$" stand for natural number equality and propositional equality, respectively. The value `tt` stands for the Boolean value *true*. Informally, the statement is read as "Given any list `xs` of elements in any set `a`, any value `ys` of `perm xs` satisfies the condition `length ys` = `length xs`, where `length` is the usual function that computes the length of a list." The proof of this claim requires a few lemmas, and is approximately one page long.

4.2 Planned choices

Our second approach encodes a Curry non-deterministic function f into a deterministic Agda function f', that takes an extra argument. This argument abstracts the non-deterministic choices made by f during the computation of a result. By passing these choices, f' becomes deterministic and executes the same steps, and produces the same result, as f.

Referring to our non-deterministic computation in the last section, the extra argument of `perm` will encode that, in line (3), the right argument of "?" has to be selected. Many proofs of non-deterministic computations involve statements that hold for each non-deterministic value, as in our previous example. Therefore, these statements are independent of the choices made during the computation. For this reason the extra argument that encodes the choices is abstract, i.e., never has a concrete value.

With this approach, we parameterize an Agda module with four abstractions:

1. `Choice` : `Set` – a type abstracting non-determinism. A value of this type plans the choices that will be made by a non-deterministic computation.
2. `choose` : `Choice` $\rightarrow \mathbb{B}$ – a function that returns whether the left or right argument of "?" should be selected.
3. `lchoice` : `Choice` $\rightarrow$ `Choice` – a function that produces choices from choices. The intent is that the choices of the result are independent of the choices of the argument. The reason for this function will be discussed shortly.
4. `rchoice` : `Choice` $\rightarrow$ `Choice` – a function like the previous one, but it produces other independent choices.

With this machinery, the function that computes a permutation of a list is defined as follows:

```
ndinsert : {A : Set} → Choice → A → 𝕃 A → 𝕃 A
ndinsert _  n []       = n :: []
ndinsert ch n (x :: xs) = if choose ch then n :: x :: xs
                                       else x :: ndinsert (lchoice ch) n xs

perm : {A : Set} → Choice → 𝕃 A → 𝕃 A
perm _  []        = []
perm ch (y :: ys) = ndinsert (lchoice ch) y (perm (rchoice ch) ys)
```

The first required argument of these functions is the plan of the non-deterministic choices. For example, in the second rule of `ndinsert`, this argument determines whether or not argument `x` should be inserted at the front of the second argument.

The second rule of `perm` justifies the necessity of functions `lchoice` and `rchoice`. Both `perm` and `ndinsert` make non-deterministic choices. The choices made by one function must be independent of the other, otherwise some intended result could be lost. Hence we need to "fork" the choices encoded by argument `ch`.

With this infrastructure, the statement that any permutation of a list L is as long as L is formalized as follows:

```
perm-length : {A : Set} → (ch : Choice) → (xs : 𝕃 A)
            → length (perm ch xs) =ℕ length xs ≡ tt
```

Informally, the statement is read as "Given any plan of non-deterministic choices `ch`, and any list `xs` of elements in any set `a`, the length of the permutation of `xs` according to `ch` is the same as the length of `xs`." The proof is only a few lines long.

5 Libraries

In order to support the direct translation of Curry programs to Agda, and to prove theorems about the translated programs, we developed Agda libraries containing some ubiquitous functions and theorems. In particular, the encoding presented in Sect. 4.1 demands such a support. Therefore, we consider only this encoding in this section.

Any Curry function can be non-deterministic. Hence, in our "set of values" encoding, any translated function has a result of type `ND` and must accept arguments of type `ND`. Thus, a Curry function of type

$\tau_1 \;\rightarrow\; \cdots \;\rightarrow\; \tau_n \;\rightarrow\; \tau$

should be translated into an Agda function of type

`ND` $\tau_1 \;\rightarrow\; \cdots \;\rightarrow\;$ `ND` $\tau_n \;\rightarrow\;$ `ND` τ

Since the type `ND` has two constructors, a direct definition of these Agda functions would be lengthy, and would also results in lengthy and tedious proofs. We can simplify the translation based on the following observations:

- Some definitional cases are identical for all functions. An argument of the form t_1 `??` t_2 leads to a non-deterministic result. The function is applied to both alternatives t_1 and t_2 and their results are combined with "`??`" (this is also called a "pull-tab" step in [3]).
- Curry functions are defined with patterns in the left-hand side. It should be sufficient to define the translated Agda functions on patterns, and use the `ND` type only in the right-hand side.

Therefore, we translate a Curry function of the type shown above into an Agda function of type

$\tau_1 \;\rightarrow\; \cdots \;\rightarrow\; \tau_n \;\rightarrow\;$ `ND` τ

which allows a more direct translation. We can even improve this translation by analyzing the operation in more detail. If the function is deterministic, i.e., it does not execute non-deterministic choices and calls only deterministic functions, we can omit the `ND` type in the result. This gives us a one-to-one correspondence between deterministic Curry functions and their Agda translation.

However, if constructors or functions are applied to other non-deterministic functions in the right-hand side of a defining rule, this application is no longer directly possible, because they do not accept arguments of type `ND`. Therefore, our Agda library provides two operations to extend such application into a non-deterministic context:

1. In order to apply a constructor or deterministic function to a non-deterministic expression, we define the application operator "`$*`":

   ```
   _$*_ : {A B : Set} → (A → B) → ND A → ND B
   f $* (Val xs) = Val (f xs)
   f $* (t1 ?? t2) = f $* t1 ?? f $* t2
   ```

   Thus, "`$*`" applies a deterministic function to all values (first rule) and distributes a non-deterministic choice to non-deterministic results (second rule). An application of "`$*`" was shown in the translation of `ndinsert` in Sect. 4.1.

2. To apply a non-deterministic function to a non-deterministic expression, we define the function "`*$*`":

   ```
   _*$*_ : {A B : Set} → (A → ND B) → ND A → ND B
   f *$* (Val x) = f x
   f *$* (t1 ?? t2) = f *$* t1 ?? f *$* t2
   ```

   The behavior on choices is similar to "`$*`". However, for each value `x`, the non-deterministic function `f` is applied to obtain a non-deterministic value (first rule). Therefore, "`*$*`" is used to combine nested calls of non-deterministic functions. The extension to non-deterministic functions of arities greater than one can be obtained by nested applications of this operator. An example application was shown in the translation of `perm` in Sect. 4.1.

In addition to these functions that support the translation of Curry functions into Agda functions, our library also includes various operations on the `ND` type. These operations support the formulation of theorems about the translated Curry programs in Agda. In general, we are interested in proving properties that are satisfied by all computed values, like the length property of permutations shown in Sect. 4.1. For this purpose, we define a predicate `satisfy` (as an infix operator) which is true if all values in a set satisfy a given predicate:

```
_satisfy_  : {A : Set} → ND A → (A → 𝔹) → 𝔹
(Val n)    satisfy p = p n
(t1 ?? t2) satisfy p = t1 satisfy p && t2 satisfy p
```

The use of this predicate to formulate a property about the non-deterministic `perm` function was shown in the statement `perm-length` in Sect. 4.1.

When one tries to prove such statements, one often needs to prove lemmas about properties of the involved functions, like "`$*`" and "`*$*`". In order to simplify these proofs, we also defined and proved a library of Agda theorems to relate these functions with predicates like `satisfy`. For instance, the following lemma relates "`$*`" and the predicate `satisfy` (where ∘ denotes function composition, i.e., `(f ∘ g) x ≡ f (g x)`):

```
satisfy-$* : {A B : Set} → (f : A → B) (xs : ND A) (p : B → 𝔹)
           → (f $* xs) satisfy p ≡ xs satisfy (p ∘ f)
```

This lemma allows us to eliminate a call to "`$*`" by moving the applied function `f` inside the predicate. The proof of this lemma is easily done by induction on the tree structure `ND`.

The formulation of a similar lemma for "`*$*`" is a bit more involved since the applied function is non-deterministic. However, if we assume the existence of a proof that this function already satisfies the predicate for all values (given as the last argument to this lemma), we can eliminate a call to "`*$*`":

```
satisfy-*$* : {A B : Set} → (p : B → 𝔹) (f : A → ND B) (xs : ND A)
            → ((y : A) → (f y) satisfy p ≡ tt) → (f *$* xs) satisfy p ≡ tt
```

Our library contains more predicates and lemmas that are useful to state and prove specific properties of Curry programs. For instance, to show that some non-deterministic Boolean expression is always true, one can use the following predicate:

```
always : ND 𝔹 → 𝔹
always (Val b)    = b
always (t1 ?? t2) = always t1 && always t2
```

A lemma to eliminate an occurrence of "`$*`" in the context of `always` is the following:

```
always-$* : {A : Set} → (p : A → 𝔹) (xs : ND A)
          → ((y : A) → p y ≡ tt) → always (p $* xs) ≡ tt
```

It states that, if there is a proof that `p` is always true, the application of `p` to a non-deterministic value is also true. An application of this lemma is shown in the following section, where we discuss some examples to prove properties of Curry programs.

6 Examples

6.1 Example 1

In this example, we assume that natural numbers are defined in Peano representation as follows:

```
data Nat = Z | S Nat
```

The following Curry function non-deterministically yields an even and an odd number close to its input value:

```
eo :: Nat → Nat
eo n = n ? (S n)
```

To show some interesting property related to `eo`, consider the function `double`, shown in Sect. 2, and the predicate `even`:

```
even :: Nat → Bool
even Z         = True
even (S Z)     = False
even (S (S x)) = even x
```

We want to prove that, for all natural numbers *n*, `even (double (eo` *n*`))` is always `True`. Note that this property holds only for the call-time choice semantics of Curry (see Sect. 2). In the run-time choice semantics, or term rewriting, this property does not hold.

To prove this property with Agda, we have to translate the program to Agda. Since `double` and `even` are deterministic functions, they can be directly translated into Agda functions, where the constructors of the Curry program are translated into corresponding constructors already defined in Agda:

```
double : ℕ → ℕ            even : ℕ → 𝔹
double x = x + x          even zero          = tt
                          even (suc 0)       = ff
                          even (suc (suc x)) = even x
```

Since non-determinism is not involved here, we can prove that `even (double` *n*`)` is always true by standard techniques. For this purpose, we state the lemma

```
even-x+x : (x : ℕ) → even (x + x) ≡ tt
```

which is immediately proved by induction on its argument. Exploiting this lemma, we can prove our intended property:

```
even-double-true : (x : ℕ) → even (double x) ≡ tt
even-double-true x rewrite even-x+x x = refl
```

In order to prove our initial property about the expression `even (double (eo` *n*`))`, we have to decide about the representation of non-determinism introduced in Sect. 4. If we choose the “set of values” representation, the function `eo` is defined as follows:

```
eo : ℕ → ND ℕ
eo n = Val n ?? Val (suc n)
```

Using the library functions presented in the previous section, the Curry expression `even (double (eo` *n*`))` is represented in Agda as:

```
(even ∘ double) $* (eo n)
```

The proof of our property is just an application of the lemma `always-$*` (see Sect. 5) to the previous lemma:

```
even-double-eo-true : (n : ℕ) → always ($* (even ∘ double) (eo n)) ≡ tt
even-double-eo-true n = always-$* (even ∘ double) (eo n) even-double-true
```

The second "choice" representation of non-determinism provides for a more direct proof. We pass the plan of non-deterministic choices as the first argument to `eo`:

```
eo : Choice → ℕ → ℕ
eo n = if choose ch then n else (suc n)
```

Then the proof is an immediate application of the lemma `even-double-true`:

```
even-double-eo-true : (ch: Choice) (n : ℕ) → even (double (eo ch n)) ≡ tt
even-double-eo-true ch n = even-double-true (eo ch n)
```

As we can see, the proof with the "choice" representation is more direct, and shorter, than with the "set of values" representation, as already mentioned in Sect. 4 for the property of the length of permutations.

6.2 Example 2

So far we have proved properties about predicates, however, we have not discussed any notion of equality. In Agda, two terms are equal if their values are syntactically equal. This is denoted by the standard definition:

```
data _≡_ {A : Set} (x : A) : A → Set where
  refl : x ≡ x
```

e.g., 1+1 ≡ 2.

However, we cannot apply this relation to non-deterministic values for the simple reason that deterministic values and non-deterministic values are structurally different. For example, given the non-deterministic value:

```
coin = (Val tt) ?? (Val ff)
```

it would seem reasonable to state that `tt ≡ coin`, but `coin` reduces to `(Val tt) ?? (Val ff)` which is not the same value as `tt`.

Instead, we must define a new notion of equality for non-deterministic values. We chose an existential interpretation of equality for our representation. If a value *x* exists inside the set of values of a non-deterministic expression *nx*, then the two are considered equivalent (non-deterministically equal). A proof of equivalence is then reduced to finding a path through the tree representing *nx*. This leads to the following structure for proofs of equivalence.

```
data _∈_ {A : Set} (x : A) : (nx : ND A) → Set where
  ndrefl : x ∈ (Val x)
  left   : (l : ND A) → (r : ND A) → x ∈ l → x ∈ (l ?? r)
  right  : (l : ND A) → (r : ND A) → x ∈ r → x ∈ (l ?? r)
```

Now we can give a proof that `tt` is in `coin`

```
hInCoin : tt ∈ coin
hInCoin = left (Val tt) (Val ff) ndrefl
```

This can be read as `tt` is on the left side of the tree `(Val tt) ?? (Val ff)`.

Now that we have a notion of equivalence, we can prove some more interesting results. Recall that "`$*`" applies a function `f` to every value of a non-deterministic expression *nx*. For consistency, it would be good to prove that if `x ∈ nx` then `f x ∈ f $* nx`. This is a simple structural induction.

```
∈-$* : {A B : Set} → (f : A → B) → (x : A) → (nx : ND A)
     → x ∈ nx → f x ∈ f $* nx

∈-$* f x (Val .x) ndrefl = ndrefl
∈-$* f x (l ?? r) (left  .l .r k) = left  (f $* l) (f $* r) (∈-$* f x l k)
∈-$* f x (l ?? r) (right .l .r k) = right (f $* l) (f $* r) (∈-$* f x r k)
```

We can prove a similar result for "`*$*`". The proof is nearly identical to `∈-$*`, but it has the general type:

```
∈-*$* : {A B : Set} → (x : A) → (nx : ND A) → (f : A → B)
      → (nf : A → ND B) → x ∈ nx → f x ∈ nf x → f x ∈ nf *$* nx
```

For a more interesting example, consider the problem of sorting a list. The problem of determining if a list is sorted has been studied extensively, but we would like to look at a conceptually more difficult problem. How can we determine if a sorted list is actually a permutation of the original list?

This has also been studied before now, and the solutions tend to be long and difficult to follow. A search of "github," "agda" and "sort" gives half a dozen independent efforts. Since we defined the non-deterministic permutation function above, it would be nice to have a theorem like (for the sake of simplicity, we consider sorting natural numbers):

```
sortPerm : (xs : 𝕃 ℕ) → sort xs ∈ perm xs
```

This is short, simple, and obviously correct if we trust `perm` to produce only permutations of a list. For this proof we will be using the following simple insertion sort.

```
insert : ℕ → 𝕃 ℕ → 𝕃 ℕ
insert x []        = x :: []
insert x (y :: ys) = if x < y then (x :: y :: ys)
                              else (y :: insert x ys)

sort : 𝕃 ℕ → 𝕃 ℕ
sort []        = []
sort (x :: xs) = insert x (sort xs)
```

We only need two lemmas. The first states that non-deterministic equivalence holds over conditional statements. This result allows us to introduce conditional statements when talking about equivalence.

```
ifIntro : {A : Set} → (x : A) → (y : A) → (nx : ND A)
        → x ∈ nx → y ∈ nx → (c : 𝔹) → (if c then x else y) ∈ nx
```

The second lemma is an equivalence between our `insert` function and the non-deterministic `ndinsert` defined above. This will give the intuitive result that, for any list `xs`, inserting a new element does not violate our non-deterministic equivalence. The type of this lemma is given by:

```
insert=ndinsert : (y : ℕ) → (xs : 𝕃 ℕ) → insert y xs ∈ ndinsert y xs
```

The proof of this lemma contains two trivial cases, but the final case gives some insight into working with non-deterministic values. The idea is straightforward. Either the value `y` is inserted at the front of the list, or it is inserted somewhere in the tail. Since our non-deterministic `ndinsert` covers both of those cases we do not need to know exactly where it is inserted. We just pick the correct case.

The full code is given below for completeness. Variables `l` and `r` represent the left and right branches in `insert` respectively, while `nl` and `nr` are the corresponding branches for `ndinsert`. Additionally, `rec` is a recursive case if `y` is inserted in the tail of the list, Finally `l∈step` and `r∈step` are proofs that `l` and `r` are somewhere in the result of inserting `y` into the list.

```
insert=ndinsert y (x :: xs) = ifIntro l r step l∈step r∈step (y < x) where
  step   = ndinsert y (x :: xs)
  l      = (y :: x :: xs)
  r      = x :: insert y xs
```

```
    nl     = Val (y :: x :: xs)
    nr     = (_::_ x) $* (ndinsert y xs)
    rec    = ∈-$* (_::_ x) (insert y xs) (ndinsert y xs) (insert=ndinsert y xs)
    l∈step = left nl nr ndrefl
    r∈step = right nl nr rec
```

Finally, we have the tools to prove our sorting theorem. After the previous result the theorem is remarkably simple.

```
sortPerm : (xs : 𝕃 ℕ) → sort xs ∈ perm xs
sortPerm [] = ndrefl
sortPerm (x :: xs) = ∈-*$* (sort xs) (perm xs) (insert x) (ndinsert x)
                           (sortPerm xs) (insert=ndinsert x (sort xs))
```

7 Partial Functions

Up to now we have considered non-deterministic but totally defined functions in our examples. However, functions in Curry can also be partially defined, and failures caused by such partial definitions are not a run-time error as in purely functional programming but a desirable feature that frees the programmer from specifying all computation details in a program. For instance, consider the problem of selecting a minimum element in a list of numbers. Such a function is easily defined in Curry:

```
minND :: [Int]  →  Int
minND xs@(_ ++ [x] ++ _) | all (x<=) xs = x
```

To specify the selection of an arbitrary element without a concrete strategy, we use a functional pattern [4] in the left-hand side of `minND`. The condition restricts the application of the rule to all selected elements that are less than or equal to all other elements. Thus, the condition fails on elements that are not minimal.

Although computations with failing branches are an important feature of functional logic languages, the modeling of failures in Agda is not straightforward. First of all, Agda requires that all functions are totally defined. Typically, the application of partially defined functions, like `head` or `tail` on lists, is restricted by a refined type, e.g., by requiring a proof that the argument of `head` or `tail` is a non-empty list. In our case this is not a reasonable solution since a Curry program actually deals with failed computations.

Another alternative to deal with partially defined functions is to extend the result type to indicate a non-successful application of a function. For instance, one can use the `Maybe` type to define `head` as a total function in Agda:

```
head : {A : Set} → 𝕃 A → maybe A
head (x :: _) = just x
head []       = nothing
```

Unfortunately, this modeling is limited to “top-level” failures. For instance, doing the same for `tail` would inhibit nested applications of `tail` due to typing reasons.

Another, and more serious, problem is the fact that Curry, as a lazy language, can also ignore failures if they occur in arguments that are not demanded. For instance, the Curry expression

```
head (0 : tail [])
```

evaluates to `0` since the failure, which occurs if `tail []` is evaluated, is ignored thanks to laziness. On the other hand, in Agda one cannot construct a list containing failures in the tail. To implement a similar expression in Agda, one has to evaluate, before constructing the list, the tail `tail []`, which returns `nothing`, so that the entire computation fails. This has the consequence that, if we prove in Agda some property about *all* successful computations, it might not be transferable to Curry, because Curry might produce additional successful computations.

A possible solution to this problem could be a different modeling of Curry's data structures in Agda. For instance, instead of mapping Curry lists to Agda lists, one could include failures and non-deterministic choices as list constructors:

```
data List (A : Set) : Set where
  empty     : List A
  cons      : A → List A → List A
  Fail-List : List A
  _??-List_ : List A → List A → List A
```

Then one can completely define `tail` on this structure:

```
tail : {A : Set } → List A → List A
tail (cons _ xs)      = xs
tail empty            = Fail-List
tail Fail-List        = Fail-List
tail (l1 ??-List l2) = tail l1 ??-List tail l2
```

With such a representation, we can "totalize" all functions, and allow failures inside data structures. Basically, it is quite similar to the translation of Curry programs into Haskell, which is the basis of the Curry implementation KiCS2 [10]. However, it is also known that this translation models the run-time choice semantics instead of the call-time choice semantics used in Curry (see Sect. 2). To obtain a call-time choice semantics, one can attach unique identifiers to each choice constructor in order to make consistent selections of computed values (see [10] for details). Although this is a systematic method to translate Curry programs into purely functional programs, it does not seem adequate for Agda since the generation of unique identifiers requires infinite data structures, and the resulting code is quite complex, requiring more complex proofs.

Although the handling of partially defined functions is difficult in general, one can still use Agda to prove properties of such functions, if the failures occur at the computation's top-level so that one can reason about their occurrences. For instance, consider the well-known functional logic definition of computing the last element of a list by exploiting the list concatenation "++":

```
last :: [a]  →  a
last xs | ys ++ [x] == xs  = x   where ys,x free
```

We want to show that `last` is a deterministic operation, i.e., for a fixed input list *xs*, the evaluation of `last` *xs* cannot yield two different results. Although the definition of `last` is partial, and uses free variables and non-determinism, the formulation and proof of this property is not difficult in Agda by considering only the condition of `last`. We show that, if there are two pairs of values for `ys` and `x` that satisfy the condition for some fixed `xs`, the two values for `x` are identical. This property can be expressed (with some type specialization to increase readability) in Agda as follows:

```
last-det : (ys1 ys2 : 𝕃 ℕ) (x1 x2 : ℕ)
         → =𝕃 _=ℕ_ (ys1 ++ [ x1 ]) (ys2 ++ [ x2 ]) ≡ tt → x1 =ℕ x2 ≡ tt
```

As a further example, consider again the non-deterministic and partial function `minND` from the beginning of this section. We want to implement the same functionality in a deterministic manner by a list traversal:

```
minD :: [Int]  →  Int
minD [x]      = x
minD (x:y:xs) = let z = minD (y:xs) in if x <= z then x else z
```

Since the correctness of this list traversal is not obvious, we take the definition of `minND` as the specification and prove that `minD` satisfies the specification, a typical task in (declarative) program development [8]. If we model `minND` with the "planned choices" translation, this property could be formulated in Agda as follows (the additional argument in the Agda function `minD` is a proof that the argument list is not empty; this is required since `minD` is partially defined but Agda accepts only total functions):

```
minlist-theorem : (ch : Choice) (x : ℕ) (xs : 𝕃 ℕ) (z : ℕ)
              → minND ch (x :: xs) ≡ just z → z =ℕ minD (x :: xs) refl ≡ tt
```

We use a `maybe` result type to express the partiality of the non-deterministic function `minND` and a proof of non-emptiness for the partiality of the deterministic function `minD` (as discussed above). Then the theorem states that, if `z` is a minimal value according to the specification, it will be computed by the deterministic implementation. The proof is non-trivial and requires various lemmas. In particular, the proof is split into showing that the non-deterministically selected element is smaller than, or equal to, than all other list elements, but it cannot be strictly smaller than all others since it is itself an element from the list.

8 Semantics

Since we prove properties of Agda functions, but are interested in properties of Curry functions, we discuss how to migrate from one environment to the other.

Initially, we consider datatypes, such as algebraically defined natural numbers, lists and trees, and functions operating on these types that have equal definitions, except for minor syntactic differences, in the two languages. We call the set of such types and functions the *intersection* of the two languages. For example, a function such as "length of a list", returning a natural number, is in the intersection. Agda functions, including those in the intersection, are totally defined, terminating, and evaluated eagerly. The rules defining these functions are unconditional, deterministic, and without free variables. Any expression in the intersection has a value, because it is an Agda expression, and this value is the same in the two languages, because in this situation eager and lazy evaluations produce the same result. Thus, our first claim is that any statement about the input/output behavior of a function, i.e., its functional correctness, proved in Agda holds for Curry.

Then, we define Agda functions that simulate non-deterministic Curry functions. For example, a function such as "permutation of a list" simulates the corresponding non-deterministic Curry function. Regardless of the approach, either *set of values* or *planned choices*, these Agda functions are still in the intersection of the two languages. Thus, we can compare a simulated non-deterministic function (in Curry and Agda the behavior is the same) against the same native non-deterministic Curry function. Of course, the expectation is that the simulated and the native functions have the same behavior. Proving this fact would not involve Agda.

Curry has a rich syntax, and some high-level constructs that are convenient for programming, and are not in the intersection with Agda. These constructs can sometimes be mapped to more elementary Curry constructs that are in the intersection of the two languages, hence the set of Curry functions that we can reason about becomes larger. For example, Curry allows free variables in expressions that are instantiated by narrowing [27, 28]. Free variables are replaced by generators [5], non-deterministic functions, that are the focus of our work. Generators are typically non-terminating functions, shortly we will discuss a workaround. Curry also allows conditional rules. With some care, the conditions of conditional rules are moved to the right-hand sides [2]. Care is needed since Agda disallows incompletely defined functions. When there are conditions it may be difficult, or impossible, to tell whether a function is defined for all inputs. Finally, Curry allows functions that do not terminate because of infinite recursion. When one such function is applied in a computation, the lazy evaluation strategy makes a recursive call only when the result of the call is demanded. To avoid non-termination in Agda, we add an additional argument that limits the number of recursive calls.

9 Related Work

There are only a few works on verifying properties of functional logic programs. One reason may be the fact that declarative programs are often considered as clear specifications so that there is nothing to prove about them. However, if one develops a more efficient version of a declarative program, because the initial version is not efficient enough in a particular application context, one wants to prove that the new version behaves like the initial version. This has been formalized in [8], where the notions of specifications and contracts for functional logic programs have been introduced together with some examples and proof obligations. Concrete methods to prove such obligations were not discussed. We have shown in this paper that it is possible to use Agda for this purpose, e.g., by proving that insertion sort computes a permutation of the input list, or that a non-deterministic selection of a minimum of a list is equivalent to a deterministic selection algorithm.

The use of proof assistants to verify functional logic programs has been pioneered in [12], where the call-time choice semantics has been formalized with different proof tools. Although the authors formally justified their translation by mapping the CRWL rewriting logic [16] into logic programs, only trivial properties, like `0 ∈ double (0?1)`, are proved. A similar approach is used in [14] where CRWL is translated into the specification language Maude, and some non-trivial but (compared to our approach) still simple properties are proved.

Since the formalization of the call-time choice semantics causes complex proofs, the proof method has been improved in [13] by a characterization of deterministic functions and expressions, and by using specific proof techniques for them. Although the authors were able to construct short proofs for deterministic properties inside functional logic programs, like the commutativity of addition, proofs for more complex properties have not been reported. These CRWL-translation based approaches can deal with the full class of functional logic programs, whereas our limited framework is able to verify non-trivial properties.

10 Conclusion

We have used the Agda system to verify the functional correctness of Curry functions. Our focus has been on Curry's non-deterministic functions, a device akin to a function that returns one of many values for the same input. Such a device contributes the logic component of the language in a lazy functional setting.

We modeled non-determinism in two different ways. One considers at once all the non-deterministic values that a function may return. The other adds an argument to a function that encodes every non-deterministic choice that the function may make. Both models eliminate the non-determinism of a functions and enable us to use Agda, which is deterministic, to prove properties of non-deterministic functions.

We developed some libraries that define non-deterministic structures and prove generic lemmas about them. With the help of these libraries, we proved some interesting properties of Curry programs. We highlighted the semantic differences between Agda and Curry, and discussed why, despite these differences, theorems proved in Agda migrate to Curry.

Proving properties of programs is a difficult task. Our effort shows that proving properties of Curry programs with Adga is possible, and that the logic component of Curry is not a major obstacle.

References

[1] S. Antoy (1997): *Optimal Non-Deterministic Functional Logic Computations*. In: *6th Int'l Conf. on Algebraic and Logic Programming (ALP'97)*, 1298, Springer LNCS, Southampton, UK, pp. 16–30, doi:10.1007/BFb0027000.

[2] S. Antoy (2001): *Constructor-based Conditional Narrowing*. In: *Proc. of the 3rd International Conference on Principles and Practice of Declarative Programming (PPDP'01)*, ACM, Florence, Italy, pp. 199–206, doi:10.1145/773184.773205.

[3] S. Antoy (2011): *On the Correctness of Pull-Tabbing*. *TPLP* 11(4-5), pp. 713–730. Available at `http://dx.doi.org/10.1017/S1471068411000263`.

[4] S. Antoy & M. Hanus (2005): *Declarative Programming with Function Patterns*. In: *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, Springer LNCS 3901, pp. 6–22, doi:10.1007/11680093_2.

[5] S. Antoy & M. Hanus (2006): *Overlapping Rules and Logic Variables in Functional Logic Programs*. In: *22nd International Conference on Logic Programming*, Springer LNCS 4079, Seattle, WA, pp. 87–101, doi:10.1007/11799573_9.

[6] S. Antoy & M. Hanus (2010): *Functional Logic Programming*. *Communications of the ACM* 53(4), pp. 74–85, doi:10.1145/1721654.1721675.

[7] S. Antoy & M. Hanus (2011): *New Functional Logic Design Patterns*. In: *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, Springer LNCS 6816, pp. 19–34, doi:10.1007/978-3-642-22531-4_2.

[8] S. Antoy & M. Hanus (2012): *Contracts and Specifications for Functional Logic Programming*. In: *Proc. of the 14th International Symposium on Practical Aspects of Declarative Languages (PADL 2012)*, Springer LNCS 7149, pp. 33–47, doi:10.1007/978-3-642-27694-1_4.

[9] Ana Bove, Peter Dybjer & Ulf Norell (2009): *A Brief Overview of Agda — A Functional Language with Dependent Types*. In: *Proceedings of the 22nd Int. Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, Springer-Verlag, Berlin, Heidelberg, pp. 73–78, doi:10.1007/978-3-642-03359-9_6.

[10] B. Braßel, M. Hanus, B. Peemöller & F. Reck (2011): *KiCS2: A New Compiler from Curry to Haskell*. In: *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*, Springer LNCS 6816, pp. 1–18, doi:10.1007/978-3-642-22531-4_1.

[11] Edmund M. Clarke & Jeannette M. Wing (1996): *Formal Methods: State of the Art and Future Directions*. *ACM Comput. Surv.* 28(4), pp. 626–643, doi:10.1145/242223.242257.

[12] J.M. Cleva, J. Leach & F.J. López-Fraguas (2004): *A logic programming approach to the verification of functional-logic programs*. In: *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, ACM Press, pp. 9–19, doi:10.1145/1013963.1013969.

[13] J.M. Cleva & F.J. López-Fraguas (2007): *Semantic Determinism and Functional Logic Program Properties*. *Electr. Notes Theor. Comput. Sci.* 174(1), pp. 3–15, doi:10.1016/j.entcs.2006.10.018.

[14] J.M. Cleva & I. Pita (2006): *Verification of CRWL Programs with Rewriting Logic*. *Journal of Universal Computer Science* 12(11), pp. 1594–1617, doi:10.3217/jucs-012-11-1594.

[15] H.B. Curry, J.R. Hindley & J.P. Seldin (1980): *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press. Available at `https://books.google.com/books?id=r0SRQAAACAAJ`.

[16] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas & M. Rodríguez-Artalejo (1999): *An approach to declarative programming based on a rewriting logic*. Journal of Logic Programming 40, pp. 47–87, doi:10.1016/S0743-1066(98)10029-8.

[17] M. Hanus (2013): *Functional Logic Programming: From Theory to Curry*. In: Programming Logics - Essays in Memory of Harald Ganzinger, Springer LNCS 7797, pp. 123–168, doi:10.1007/978-3-642-37651-1_6.

[18] M. Hanus (ed.) (2016): *Curry: An Integrated Functional Logic Language (Vers. 0.9.0)*. Available at `http://www.curry-language.org`.

[19] H. Hussmann (1992): *Nondeterministic Algebraic Specifications and Nonconfluent Term Rewriting*. Journal of Logic Programming 12, pp. 237–255, doi:10.1016/0743-1066(92)90026-Y.

[20] John McCarthy (1963): *A Basis for a Mathematical Theory of Computation*. In: Computer Programming and Formal Systems, North-Holland, pp. 33–70, doi:10.1016/S0049-237X(08)72018-4.

[21] Kenneth L. McMillan (1993): *Symbolic model checking*. Kluwer Academic, Boston. Available at `http://opac.inria.fr/record=b1084184`, doi:10.1007/978-1-4615-3190-6.

[22] Bertrand Meyer (1992): *Applying "Design by Contract"*. Computer 25(10), pp. 40–51, doi:10.1109/2.161279.

[23] Flemming Nielson, Hanne R. Nielson & Chris Hankin (1999): *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, doi:10.1007/978-3-662-03811-6.

[24] Ulf Norell (2009): *Dependently Typed Programming in Agda*. In: Proceedings of the 4th International Workshop on Types in Language Design and Implementation, TLDI '09, ACM, New York, NY, USA, pp. 1–2, doi:10.1145/1481861.1481862.

[25] Ulf Norell (2009): *Dependently Typed Programming in Agda*. In: Proceedings of the 6th International Conference on Advanced Functional Programming, AFP'08, Springer-Verlag, Berlin, Heidelberg, pp. 230–266. Available at `http://dl.acm.org/citation.cfm?id=1813347.1813352`, doi:10.1007/978-3-642-04652-0_5.

[26] S. Peyton Jones, editor (2003): *Haskell 98 Language and Libraries—The Revised Report*. Cambridge University Press.

[27] U. S. Reddy (1985): *Narrowing as the Operational Semantics of Functional Languages*. In: Proc. IEEE Internat. Symposium on Logic Programming, Boston, pp. 138–151.

[28] J. R. Slagle (1974): *Automated Theorem-Proving for Theories with Simplifiers Commutativity, and Associativity*. J. ACM 21(4), pp. 622–642, doi:10.1145/321850.321859.