Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs

Ana Bove[1], Peter Dybjer[1], and Andrés Sicard-Ramírez[2] ⋆

[1] Chalmers University of Technology, Gothenburg, Sweden
[2] EAFIT University, Medellín, Colombia

Abstract. We propose a new approach to the computer-assisted verification of functional programs. We work in first order theories of functional programs which are obtained by extending Aczel's first order theory of combinatory formal arithmetic with positive inductive and coinductive predicates. Rather than building a special purpose system we implement our theories in Agda, a proof assistant for dependent type theory which can be used as a generic theorem prover. Agda provides support for interactive reasoning by encoding first order theories using the formulae-as-types principle. Further support is provided by off-the-shelf automatic theorem provers for first order logic which can be called by a program which translates Agda representations of first order formulae into the TPTP language understood by the provers. We show some examples where we combine interactive and automatic reasoning, covering both proof by induction and coinduction.

1 Introduction

The goal of this paper is to show a simple way to build a system for reasoning about programs in functional languages with higher order functions, general recursion and lazy evaluation in the style of Haskell [23]. Building a mature proof assistant from scratch for this purpose is a daunting task, although there are some attempts in this direction [15,20]. Here we suggest to achieve this goal by building on existing state-of-the-art systems in interactive and automatic theorem proving. Our solution combines the following three strands of research:

- Using a logic for *general recursive* functional programs [9,10,11] which is based on Aczel's *first order theory of combinatory arithmetic* [3]; we extend this theory to deal in a seamless way with full general recursion, higher order functions, termination proofs, and inductive and coinductive predicates.
- Using *automatic theorem provers* for proving properties of functional programs by translating them into *first order logic* as proposed by Claessen and Hamon in their work on "The Cover Translator" (Chalmers, 2003).
- Using automatic theorem provers for first order logic for proof assistants based on *dependent type theory*, see Tammet and Smith's Gandalf [27], and Abel, Coquand, and Norell's AgdaLight [1].

⋆ Part of this research was performed during a research visit to Chalmers University of Technology which was funded by the ALFA network LERnet and EAFIT University.

We use the Agda system [28] as our interactive theorem prover. It is simultaneously a dependently typed functional programming language and a proof assistant. It is an extension of Martin-Löf type theory with numerous programming language features which facilitate programming and interactive proof construction.

Like Martin-Löf type theory, Agda has the strong normalisation property. This property is ensured by only allowing restricted forms of recursion. A consequence is that one cannot write programs by arbitrary general recursion. It is the goal of the *dependently typed programming* community to turn this restricted discipline of programming into a practical methodology.

In this paper we directly verify mainstream general recursive functional programs. To this end we use Agda as a *logical framework* in much the same way as the Edinburgh logical framework [14], that is, as a meta-logical system which is used as a basis for the implementation of a range of special purpose logics. Our logic is a first order theory of combinators (FOTC) based on Aczel's theory [3]. When implementing FOTC in Agda we get access to advanced features for interactively building proofs in the proof assistant, such as, commands for refining proof terms, definition by pattern matching, flexible mixfix syntax accepting Unicode, etc.

Furthermore, we provide a translation of Agda representations of formulae in the FOTC into the TPTP language [26] so that we can call off-the-shelf automatic theorem provers (ATPs) when proving properties of our programs.

A key point of our approach is that Martin-Löf type theory is a subsystem of our theory through a natural interpretation [3]. However, our theory is strictly more general; in particular, we can write arbitrary general recursive functional programs. This extra generality comes at a price: since we can now reason about programs which do not terminate, we can no longer make use of the automatic type-checking in the same way as before. To compensate for this loss we use automatic first order theorem proving, although it does not fully replace the type-checking algorithm as we shall see. On the other hand, the ATPs can prove theorems automatically which would otherwise require manual proofs.

Overview of the paper. Section 2 introduces our FOTC for Plotkin's PCF language. In Section 3 we explain how to encode first order theories in Agda and how to instruct the proof assistant to call the ATPs. Section 4 shows how to encode FOTC for PCF in Agda and how this enables us to combine interactive and automatic theorem proving. In Section 5 we extend FOTC by adding inductive and coinductive predicates and we present an example using both. Finally, Section 6 contains some discussion of future and related work.

The programs and the examples described in the paper are available at `www1.eafit.edu.co/asicard/code/fossacs-2012/`.

2 First Order Theories of Combinators

As we mentioned before, Aczel showed how to interpret Martin-Löf type theory in traditional first order logic. He gave an *abstract realisability interpretation*, where

the *proof objects* are interpreted as terms in combinatory logic and *types* are interpreted as unary predicates. Aczel's first order theory only has two constants (K and S) and one binary function symbol (for application). This is because all the term formers of Martin-Löf type theory can be encoded in the usual way using bracket abstraction, Church encodings, and fixed point operators. The theory also has three unary predicate symbols $\mathcal{N}$, $\mathcal{P}$, and $\mathcal{T}$ meaning that a combinatory term encodes a natural number, an internal proposition, and an internal true proposition, respectively. Aczel's paper was the first of several papers on realisability interpretations of Martin-Löf type theory; see for example Aczel [4] and Smith [25].

A Logic for PCF with Totality Predicates. Dybjer [9] showed that one of these logics for realisability interpretations, the so called *Logical Theory of Constructions (LTC)* is appropriate for practical verification of functional programs. This logic is closely related to Aczel's first order theory, but is based on the λ-calculus, and is hence not a first order theory.

For the purpose of this paper we begin by considering an LTC-style logic for Plotkin's PCF language [24]. PCF does not have internal propositions, hence we do not need the predicate symbols $\mathcal{P}$ and $\mathcal{T}$. On the other hand, we have two unary predicate symbols $\mathcal{B}ool$ and $\mathcal{N}$, where $\mathcal{B}ool(t)$ means that t is a *total* boolean value (true or false), and $\mathcal{N}(t)$ that t is a *total* natural number. We will use these predicates to assert that a certain (possibly non-terminating) PCF program terminates with a total boolean value or a total natural number, respectively.

In a previous paper [7] we showed how to use Agda for implementing this LTC-style logic. The aim of the present paper is to make use of off-the-shelf automatic theorem provers for first order logic. Hence, we must make our logic first order by removing λ-abstraction. Instead, we work in an extensible theory and add a new function symbol for each recursive function definition of the form

$$f\, x_1 \cdots x_n = e[f, x_1, \ldots, x_n].$$

It is well-known how to translate such definitions into terms using λ-abstraction and fixed point operators. For convenience, we might actually define function symbols by pattern matching, whenever it is clear that this pattern matching can be replaced by a single recursive equation by using if, pred and iszero.

The grammar for terms is now first order:

$$t ::= x \mid t\, t \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if} \mid 0 \mid \mathsf{succ} \mid \mathsf{pred} \mid \mathsf{iszero} \mid f$$

where f ranges over new combinators defined by recursive equations as above. The axioms can be classified into three groups: (i) conversion rules for the combinators, (ii) discrimination rules expressing that terms beginning with different constructors are not convertible, and (iii) introduction and elimination rules for $\mathcal{B}ool$ and $\mathcal{N}$. We show these axioms in Section 4.

3 Combining Interactive and Automatic Proofs in First Order Logic

3.1 First Order Logic in Agda

The encoding of intuitionistic first order logic in dependent type theory using the formulae-as-types principle is of course well-known; below we briefly show what it looks like in Agda. For example, to implement disjunction we encode it as the disjoint union; note that below, we declare the constants as *postulates*.

```
postulate _∨_ : Set → Set → Set
          inl : {A B : Set} → A → A ∨ B
          inr : {A B : Set} → B → A ∨ B
          case : {A B C : Set} → (A → C) → (B → C) → A ∨ B → C
```

The first constant declares the syntax of disjunction as an infix binary set former. The second and third constants declare the introduction rules, and the fourth the elimination rule. Note that these rules are *axiom schemata* that is, they are sets of first order formulae, one for each instance of `A`, `B` and `C`. Agda is a *higher order* logic; to express the schematic nature of these rules we use (implicit) quantification over `Set`. Curly brackets {,} declare *implicit* arguments, that is, arguments that do not appear explicitly in the proof terms.

The proof of commutativity of disjunction can now be written as

```
commOr : {A B : Set} → A ∨ B → B ∨ A
commOr c = case inr inl c
```

By using postulates we can encode all of classical first order logic. The *adequacy problem* —the question of whether such an encoding gives rise to exactly the same provable formulae as the original theory— is studied by Gardner [12].

However, to make the most of the proof assistant it is preferable to use Agda's *data* declarations for inductively defined types, whenever appropriate. Hence, we declare the syntax and the introduction rules for disjunction as follows:

```
data _∨_ (A B : Set) : Set where
  inl : A → A ∨ B
  inr : B → A ∨ B
```

We can now write proofs by pattern matching; for example the proof of commutativity of disjunction becomes

```
commOr : {A B : Set} → A ∨ B → B ∨ A
commOr (inl a) = inr a
commOr (inr b) = inl b
```

When we encode our theory using `data` rather than `postulate` we get a new adequacy problem, since we have a more general language where we can write proofs by pattern matching. Here, we should only use pattern matching in ways which are reducible to the `case` combinator, encoding disjunction elimination.

We shall use `data` for all logical constants, the equality relation (denoted as ≡), and the totality predicates in our FOTC (with the same remark as above).

Furthermore, to define the quantifiers we postulate a domain of individuals:

```
postulate D : Set
```

The *universal quantifier* is implemented by the dependent function type `(x : D) → P`. If the domain `D` can be deduced by the type checker, we use the alternative notation `∀ x → P` for this type.

Finally, since the automatic theorem provers implement *classical* first order logic we need to include (a postulate for) the law of excluded middle:

```
postulate lem : {A : Set} → A ∨ ¬ A
```

3.2 Combining Agda with Automatic Theorem Provers

We have modified Agda by adding *pragmas* containing information to be used by the ATPs. These pragmas instruct the system to add information in an interface file which is generated after type-checking a file. In this way we tell the ATPs to prove a certain formula, or that a certain formula is an axiom or a general hint, or that a certain constant is a definition.

We tell the ATPs that the formula `name` is an axiom by the pragma

```
{-# ATP axiom name #-}
```

To prove a property automatically we first postulate it and add the pragma that instructs the ATPs to prove this conjecture. For example, to prove commutativity of disjunction automatically we write

```
postulate commOr : {A B : Set} → A ∨ B → B ∨ A
{-# ATP prove commOr #-}
```

After type-checking we run the program `agda2atp`, which first translates all axioms, definitions and conjectures in the generated interface file into the TPTP language, and then tries to prove the conjectures calling independently the automatic theorem provers E, Equinox, SPASS, Metis, or Vampire. In the terminal, we get information about which property is being proved and which ATP was able to prove a property first, if any.

```
Proving the conjecture in /tmp/Examples.commOr_7.tptp ...
Vampire 0.6 (...) proved the conjecture in /tmp/Examples.commOr_7.tptp
```

If no ATP could prove a conjecture within five minutes (by default), the process is cancelled and the ATPs will continue and try to prove the next conjecture.

It is possible to specify local hints in the pragma {`-# ATP prove ... #-`} by giving their names after the name of the conjecture to be proved.

4 Implementing FOTC for PCF in Agda

We first declare the syntax of PCF terms as the following postulates:

```
postulate if_then_else_ : D → D → D → D
          _·_ : D → D → D
          succ pred isZero : D → D
          zero true false : D
```

Note that if we were faithful to the syntax of PCF given in Section 2, `if`, `succ`, `pred` and `isZero` would have type `D`. However, the above versions are definable, and easier to use with the theorem prover (and easier to read for humans).

We now postulate the conversion rules, and add a pragma which declare them to be axioms for the ATPs:

```
postulate if-true : ∀ d₁ {d₂} → if true then d₁ else d₂ ≡ d₁
          if-false : ∀ {d₁} d₂ → if false then d₁ else d₂ ≡ d₂
          pred-S : ∀ d → pred (succ d) ≡ d
          isZero-0 : isZero zero ≡ true
          isZero-S : ∀ d → isZero (succ d) ≡ false
{-# ATP axiom if-true if-false pred-S isZero-0 isZero-S #-}
```

We omit the discrimination rules.

Then we define a predicate for total natural numbers as a data type, and the induction schema for natural numbers by pattern matching:

```
data N : D → Set where
  zN : N zero
  sN : ∀ {n} → N n → N (succ n)
{-# ATP axiom zN sN #-}

indN : (P : D → Set) → P zero →
       (∀ {n} → P n → P (succ n)) → ∀ {n} → N n → P n
indN P P0 h zN = P0
indN P P0 h (sN Nn) = h (indN P P0 h Nn)
```

Note that since induction is a *schema* we cannot declare it as an axiom until it is instantiated. There are analogous rules for total Booleans.

Let us now add a combinator for addition. We postulate a binary infix operation on `D` and the (recursive) equations as axioms for the ATPs.

```
postulate _+_ : D → D → D
          +-0x : ∀ d → zero + e ≡ e
          +-Sx : ∀ d e → succ d + e ≡ succ (d + e)
{-# ATP axiom +-0x +-Sx #-}
```

We can show that addition is a total function on natural numbers by induction on the first argument. If we manually instantiate the induction schema, then both cases can be proved automatically (using a hint in the proof of $\texttt{+-N}_1$):

```
indN-instance : ∀ x → N (zero + x) →
                (∀ {n} → N (n + x) → N (succ n + x)) →
                ∀ {n} → N n → N (n + x)
indN-instance x = indN (λ i → N (i + x))

postulate +-N₁ : ∀ {m n} → N m → N n → N (m + n)
{-# ATP prove +-N₁ indN-instance #-}
```

A more convenient way to instantiate the induction schema is to instruct Agda to do pattern matching on the first argument:

```
+-N : ∀ {m n} → N m → N n → N (m + n)
+-N {n = n} zN Nn = prf
  where postulate prf : N (zero + n)
        {-# ATP prove prf #-}
+-N {n = n} (sN {m} Nm) Nn = prf (+-N Nm Nn)
  where postulate prf : N (m + n) → N (succ m + n)
        {-# ATP prove prf #-}
```

To prove commutativity of addition we proceed in the same way: we do pattern matching on one of the arguments, then we prove the base case and the step case of the induction automatically.

```
+-comm : ∀ {m n} → N m → N n → m + n ≡ n + m
+-comm {n = n} zN Nn = prf
  where postulate prf : zero + n ≡ n + zero
        {-# ATP prove prf +-rightIdentity #-}
+-comm {n = n} (sN {m} Nm) Nn = prf (+-comm Nm Nn)
  where postulate prf : m + n ≡ n + m → succ m + n ≡ n + succ m
        {-# ATP prove prf x+Sy≡S[x+y] #-}
```

Here we used the following hints, which both were proved automatically:

```
+-rightIdentity : ∀ {n} → N n → n + zero ≡ n
x+Sy≡S[x+y] : ∀ {m n} → N m → N n → m + succ n ≡ succ (m + n)
```

An example with nested recursion. McCarthy's 91-function is defined by the following axiom:

```
postulate mc91 : D → D
          mc91-eq : ∀ n → mc91 n ≡
                      if n > 100 then n ∸ 10 else mc91 (mc91 (n + 11))
{-# ATP axiom mc91-eq #-}
```

We shall show that it has the following property:

```
mc91-res≯100 : ∀ {n} → N n → n ≯ 100 → mc91 n ≡ 91
```

The proof is done interactively by well-founded induction on the relation `101 ∸ m < 101 ∸ n`. Most of the auxiliary properties are proved with the help of the ATPs. We show only a few of them.

First we show that `mc91 100 ≡ 91` by using the ATPs

```
 postulate mc91-res-100 : mc91 100 ≡ 91
{-# ATP prove mc91-res-100 100+11>100 100+11∸10>100
                           101≡100+11∸10 91≡100+11∸10∸10 #-}
```

where the hints are arithmetic properties which are proved automatically. To prove the remaining cases, we use a lemma that is proved automatically:

```
postulate mc91x-res≯100 : ∀ m n → m ≯ 100 → mc91 (m + 11) ≡ n →
                                    mc91 n ≡ 91 → mc91 m ≡ 91
{-# ATP prove mc91x-res≯100 #-}
```

Let `m < 100`. To compute `mc91 m` we use `mc91-eq`, for which we first need to compute `mc91 (m + 11)`. Which branch of the definition of `mc91` we use for this computation depends of the value of `m`.

If `90` $\leq$ `m` $\leq$ `99` then `m + 11 > 100`, so we apply the true-branch and obtain `(m + 11)` $\dot{-}$ `10` and we apply `mc91` again to the result of `mc91 (m + 11)`. We now use `mc91x-res`$\not>$`100` to prove that `mc91 m` returns `91`. For the case of `98` we have:

```
postulate mc91-res-109 : mc91 (98 + 11) ≡ 99
          mc91-res-99 : mc91 99 ≡ 91
{-# ATP prove mc91-res-109 98+11>100 x+11∸10≡Sx #-}
{-# ATP prove mc91-res-99 mc91x-res⊁100 mc91-res-110 mc91-res-100 #-}
```

On the other hand, if `m` $\leq$ `89` then `m + 11` $\not>$ `100`. Hence, our inductive hypothesis tells us that `mc91 (m + 11)` $\equiv$ `91`. Using `mc91x-res`$\not>$`100` on the inductive hypothesis and on the proof that `mc91 91` $\equiv$ `91` we obtain the desired result.

Additionally, using well-founded induction on the relation `101` $\dot{-}$ `m < 101` $\dot{-}$ `n` and with the help of the ATPs, we proved that `mc91` is a total function, we prove that `mc91 n` $\equiv$ `n` $\dot{-}$ `10` when `n > 100`, and we prove that $\forall$ `n. n < (mc91 n + 11)`.

5 Adding Inductive and Coinductive Predicates

5.1 Inductive Predicates

Note that FOTC for PCF is not *one* first order theory; it is a family of first order theories. When we add a new recursive function, we extend the theory with a new function symbol and one (or several) equational axioms. As we already remarked, it is easy to extend the model accordingly, since the model is based on Scott domains with a fixed point operator.

Furthermore, in addition to our inductively defined totality predicates $\mathcal{N}$ and $\mathcal{B}ool$, we may add other inductively defined predicates. For example, we may add a new inductively defined unary predicate symbol $\mathcal{E}ven$ with axioms stating the *introduction rules* that `zero` is an even number and that even numbers are closed under the function which adds 2 to a natural number; and the induction schema stating that $\mathcal{E}ven$ is the least predicate with those properties.

A schema for (intuitionistically valid) inductive predicates in first order logic is given by Martin-Löf [18]. However, since we work in classical logic, nothing prohibits us from adding inductively generated predicates by arbitrary (not necessarily strictly) positive operators, since they can easily be modelled as least fixed points of monotone operators on subsets of the domain [2].

5.2 An Example with Higher-Order Recursion

Here we define the mirror function for general trees in FOTC. First we extend our language with constructors for lists and trees:

```
postulate [] : D
          _::_  node : D → D → D
```

Then we mutually define predicates for total forests and trees.

```
mutual data Forest : D → Set where
            nilF : Forest []
            consF : ∀ {t ts} → Tree t → Forest ts → Forest (t :: ts)
       data Tree : D → Set where
            treeT : ∀ d {ts} → Forest ts → Tree (node d ts)
```

(For space reasons we will omit the pragmas instructing the ATPs about axioms.)

Furthermore, we define the `map` function for lists

```
postulate map : D → D → D
          map-[] : ∀ f → map f [] ≡ []
          map-:: : ∀ f d ds → map f (d :: ds) ≡ f · d :: map f ds
```

and the `mirror` function for trees:

```
postulate mirror : D
          mirror-eq : ∀ d ts → mirror · (node d ts) ≡
                               node d (reverse (map mirror ts))
```

We prove the following property:

```
mirror² : ∀ {t} → Tree t → mirror · (mirror · t) ≡ t
```

We do induction on the proof that the tree is total and then on its underlying forest; we obtain two cases depending on whether the forest is empty or not.

```
mirror² (treeT d nilF) = prf
  where postulate prf : mirror · (mirror · node d []) ≡ node d []
        {-# ATP prove prf #-}
mirror² (treeT d (consF {t} {ts} Tt Fts)) = prf
  where postulate prf : mirror · (mirror · node d (t :: ts)) ≡
                        node d (t :: ts)
        {-# ATP prove prf helper #-}
```

The hint `helper` is the following lemma:

```
helper : ∀ {ts} → Forest ts →
         reverse (map mirror (reverse (map mirror ts))) ≡ ts
```

It follows by induction on forests. Both cases are proved automatically.

5.3 Coinductive Predicates

We shall now show how to prove the correctness of a functional programming version of the alternating bit protocol (ABP). The purpose of this protocol is to ensure safe communication over an unreliable transmission channel. The sender tags the message with an (alternating) bit which is checked by the receiver. In the case of proper transmission the receiver sends the bit back to the sender as an acknowledgment. Otherwise, it sends the opposite bit back to signal that the message needs to be resent.

We follow Dybjer and Sander [11] who showed how to represent the ABP as a *Kahn network*, that is, as a network of communicating stream transformers, written in the lazy functional programming language Miranda [30] (a precursor of Haskell). They proved it correct in Park's μ-calculus [21]. This is an extension of first order classical logic with a μ-operator: for any positive formula $\Phi[X]$ with a free predicate variable X, we can form $\mu X.\Phi[X]$, with axioms which express that (i) $\mu X.\Phi[X]$ is a *prefixed point* of $\Phi[X]$ (the introduction rule for the *inductive predicate*), and (ii) that it is the *least prefixed point* (the elimination rule or induction principle). Since we work in classical logic we automatically have *coinductive predicates*, since *greatest fixed points* of $\Phi[X]$ can be defined as special least fixed points by dualisation.

Dybjer and Sander implemented the μ-calculus in the Isabelle system [22], and the proof was mechanically checked using Isabelle's tactics.

We here show how to modify Dybjer and Sander's approach so that it fits within first order logic. Rather than using the μ-operator (a second order construct) for inductive and coinductive predicates, we add new predicate symbols to our first order theories with axioms and axiom schemata corresponding to the least and greatest fixed point properties, respectively. In the previous section we showed how to add some inductive predicates. Now we will also add some coinductive predicates which will be used in the proof of the correctness of the alternating bit protocol.

Our first example is the coinductive definition of the predicate expressing that a certain list is infinite or *productive*. We add a unary predicate symbol `Stream` and two axioms expressing (i) that it is a *postfixed point* of a certain operator, and (ii) that it is the *greatest* such postfixed point:

```
Stream-gfp₁ : ∀ {xs} → Stream xs →
              ∃[ x' ] ∃[ xs' ] Stream xs' ∧ xs ≡ x' :: xs'
Stream-gfp₂ : (P : D → Set) →
              (∀ {xs} → P xs → ∃[ x' ] ∃[ xs' ]
                        P xs' ∧ xs ≡ x' :: xs') →
              ∀ {xs} → P xs → Stream xs
```

Similarly, we coinductively define when two streams are *bisimilar*:

```
≈-gfp₁ : ∀ {xs ys} → xs ≈ ys → ∃[ x' ] ∃[ xs' ] ∃[ ys' ] xs' ≈ ys'
         ∧ xs ≡ x' :: xs' ∧ ys ≡ x' :: ys'
≈-gfp₂ : (_R_ : D → D → Set) → (∀ {xs ys} → xs R ys →
           ∃[ x' ] ∃[ xs' ] ∃[ ys' ] xs' R ys'
           ∧ xs ≡ x' :: xs' ∧ ys ≡ x' :: ys') →
         ∀ {xs ys} → xs R ys → xs ≈ ys
```

In order to express the correctness property of the ABP we need a certain fairness property of the unreliable transmission channels. This property will be encoded in terms of oracle bit streams, where the bits `T` and `F` represent proper and improper transmission, respectively. Fairness here means that the bit stream contains an infinite number of `T`s and is defined as follows:

```
Fair-gfp₁ : ∀ {fs} → Fair fs →
            ∃[ ft ] ∃[ fs' ] F*T ft ∧ Fair fs' ∧ fs ≡ ft ++ fs'
```

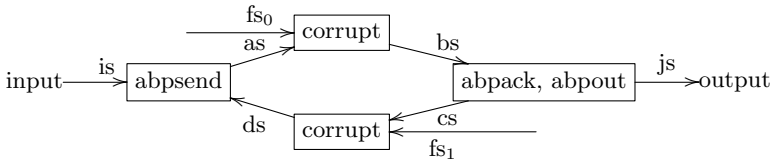

Fig. 1. The alternating bit protocol.

```
Fair-gfp₂ : (P : D → Set) → (∀ {fs} → P fs →
              ∃[ ft ] ∃[ fs' ] F*T ft ∧ P fs' ∧ fs ≡ ft ++ fs') →
            ∀ {fs} → P fs → Fair fs
```

Here `F*T ft` is an inductive predicate expressing that `ft` is a finite list of `F`s followed by a final `T`. Note that we have added the constant symbols `T` and `F` for bits, and a binary infix function symbol `++` for appending lists. In the proof below we will also make use of the predicate `Bit : D → Set`. Moreover, we use `<_,_>` for pairs, `not` for negation of bits, `error` for a corrupted message, and `ok` for a constructor for a proper message.

5.4 A Kahn Network for the Alternating Bit Protocol

Dybjer and Sander model the sender as a stream transformer `abpsend` and the receiver as a pair of stream transformers `abpack`, which returns the acknowledgement stream `cs`, and `abpout`, which returns the output stream `js`. Moreover, an unreliable transmission channel is modelled as a stream transformer, which non-deterministically corrupts the messages in the stream. To stay within the framework of deterministic lazy functional programming, we model the channels as a stream transformer `corrupt : D` which accepts an oracle stream as an auxiliary argument as described above (see Fig 1). The axioms for `corrupt` are:

```
corrupt-T : corrupt · (T :: fs) · (x :: xs) ≡ ok x :: corrupt · fs · xs
corrupt-F : corrupt · (F :: fs) · (x :: xs) ≡ error :: corrupt · fs · xs
```

(Note that for space reasons we have omitted the universal quantifiers. We will do so in the sequel as well. We will also omit the keyword `postulate`.)

The sender is written as a program which is mutually recursive with an auxiliary program `await`:

```
abpsend-eq : abpsend · b · (i :: is) · ds ≡ < i , b > :: await b i is ds
await-ok≡ : b ≡ b₀ → await b i is (ok b₀ :: ds) ≡
                       abpsend · (not b) · is · ds
await-ok≠ : ¬ (b ≡ b₀) → await b i is (ok b₀ :: ds) ≡
                          < i , b > :: await b i is ds
await-error : await b i is (error :: ds) ≡ < i , b > :: await b i is ds
```

The first order axioms for the receiver programs `abpout` and `abpack` are

```
abpack-ok≡ : b ≡ b₀ → abpack · b · (ok < i , b₀ > :: bs) ≡
                      b :: abpack · (not b) · bs
abpack-ok≢ : ¬ (b ≡ b₀) → abpack · b · (ok < i , b₀ > :: bs) ≡
                          not b :: abpack · b · bs
abpack-error : abpack · b · (error :: bs) ≡ not b :: abpack · b · bs

abpout-ok≡ : b ≡ b₀ → abpout · b · (ok < i , b₀ > :: bs) ≡
                      i :: abpout · (not b) · bs
abpout-ok≢ : ¬ (b ≡ b₀) → abpout · b · (ok < i , b₀ > :: bs) ≡
                          abpout · b · bs
abpout-error : ∀ b bs → abpout · b · (error :: bs) ≡ abpout · b · bs
```

We can now write a function `abptransfer` which computes the output `js` from the input `is`, and accepts three more arguments: the initial bit `b`, and the two oracle streams $\mathtt{os}_0$ and $\mathtt{os}_1$:

```
abptransfer-eq : abptransfer b fs₀ fs₁ is ≡
                 transfer (abpsend · b) (abpack · b) (abpout · b)
                          (corrupt · fs₀) (corrupt · fs₁) is
```

Here `transfer` is the general transfer function for the network topology of Fig. 2. It simultaneously computes the output `js` and the streams `as, bs, cs, ds` given

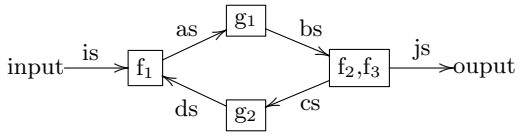

Fig. 2. Network topology for the alternating bit protocol.

the stream transformers $\mathtt{f}_1$, $\mathtt{f}_2$, $\mathtt{f}_3$, $\mathtt{g}_1$, $\mathtt{g}_2$:

```
transfer-eq : transfer f₁ f₂ f₃ g₁ g₂ is ≡ f₃ · (hbs f₁ f₂ f₃ g₁ g₂ is)
has-eq : has f₁ f₂ f₃ g₁ g₂ is ≡ f₁ · is · (hds f₁ f₂ f₃ g₁ g₂ is)
hbs-eq : hbs f₁ f₂ f₃ g₁ g₂ is ≡ g₁ · (has f₁ f₂ f₃ g₁ g₂ is)
hcs-eq : hcs f₁ f₂ f₃ g₁ g₂ is ≡ f₂ · (hbs f₁ f₂ f₃ g₁ g₂ is)
hds-eq : hds f₁ f₂ f₃ g₁ g₂ is ≡ g₂ · (hcs f₁ f₂ f₃ g₁ g₂ is)
```

To prove that the alternating bit protocol is correct means to prove that each message is eventually transmitted properly. Formally this means that the input stream is bisimilar to the output stream computed by `abptransfer`. This property can only hold if one assumes that the transmission channel(s) are "fair" in the sense described above. Formally we thus need to prove

```
spec : Bit b → Stream is → Fair fs₀ → Fair fs₁ →
       is ≈ abptransfer b fs₀ fs₁ is
```

The proof is by coinduction. We prove the `is` and `js` are in the greatest bisimulation $\_\approx\_$ by finding another bisimulation which they are in. This proof uses an auxiliary proof by induction on the predicate `F*T`.

As in the previous examples, we manually need to instantiate the axiom schemata (for induction and coinduction), but once this has been done a major part (but not all) equational and logical reasoning is done automatically by the ATPs. We do not have space here to present the details of this proof. The reader is referred to the paper's website.

6 Conclusions and Related Work

What is unique about our approach is that its logical basis is Aczel's first order theories of combinators. These theories were used for interpreting early versions of Martin-Löf's intuitionistic type theory, and our approach can be summarised by saying that we work in models of type theory rather than in type theory itself. In particular we make essential use of totality predicates (which were used for interpreting types of type theory) and other inductive definitions.

A similar viewpoint has been exploited in the NuPrl project [29], where Martin-Löf type theory is also viewed through an interpretation in untyped computation systems. The difference is that in NuPrl the user still works in an extension of (extensional) Martin-Löf's type theory, while we work in a setting which abandons most of the characteristics of Martin-Löf type theory. We work in classical rather than intuitionistic logic; we do not use the formulae-as-types principle; we have no dependent types, in fact our language is untyped rather than typed; and we deal with non-terminating as well as terminating programs. The advantage is that we can write our functional programs in the usual way as in mainstream functional languages. Although our term language is untyped, we may use polymorphic type inference during programming. However, the inferred types play no role during verification.

In this work we use the Agda system, but we would also carry out similar work using another generic theorem prover such as Isabelle. However, Agda seems to work well as an interface to automatic first order theorem provers is positive: we have used it not only for FOTC but also for other first order theories such as Group Theory and Peano Arithmetic with encouraging results.

Future research. The present approach can be improved in several ways. The most obvious is to extend Agda so that it gives more support for FOTC and for interacting with ATPs. It would also be interesting to modify our program `agda2atp` and return witnesses for the automatically generated proofs so that they can be checked by Agda. Another interesting direction is to connect Agda to systems which can automatically do proof by induction; currently we only automate pure first order logic reasoning. In fact, Agda comes with its own automatic theorem prover *Agsy - the Agda Synthesiser* which can do proof by induction [17].

Related work. There is much related work on different aspects of this topic, and we only have space to mention a few. Perhaps most importantly, we should compare our approach with other systems which can be used for reasoning about general recursive programs, going back at least to the LCF-system [13]. We already mentioned some recent dedicated such systems [15,20]. Another interesting approach is the function package [16] built on top of the Isabelle system. The logical basis is here different from ours: the basic idea is to interpret first order functions as relations in Isabelle-HOL. The function package can also deal with higher order functions. Moreover, the Boyer-Moore theorem prover [8] is a powerful system for automatically proving properties of programs by induction. Logically, however it is based on primitive recursive arithmetic rather than untyped combinatory logic as ours. Yet another system in somewhat the same spirit as ours is Schwichtenberg's Minlog [5].

We will only mention some other related areas. One such area is concerned with methods for encoding general recursive functions in intuitionistic type theory, see for example [6]. Another area is concerned with connecting theorem provers with dependent type theory [1,27] or other generic theorem provers such as Isabelle [19].

References

1. Abel, A., Coquand, T., Norell, U.: Connecting a logical framework to a first-order logic prover. In: Gramlich, B. (ed.) Proc. of 5th International Workshop on Frontiers of Combining Systems. LNAI, vol. 3717, pp. 285–301 (2005)
2. Aczel, P.: An introduction to inductive definitions. In: Barwise, J. (ed.) Handbook of Mathematical Logic, pp. 739–782. North-Holland Publishing Company (1977)
3. Aczel, P.: The strength of Martin-Löf's intuitionistic type theory with one universe. In: Miettinen, S., Väänanen, J. (eds.) Proc. of the Symposium on Mathematical Logic (Oulu, 1974). pp. 1–32. Report No. 2, Department of Philosopy, University of Helsinki, Helsinki (1977)
4. Aczel, P.: Frege structures and the notions of proposition, truth and set. In: Barwise, J., et al. (eds.) The Kleene Symposium, pp. 31–59. Amsterdan: North-Holland (1980)
5. Benl, H., et al.: Proof theory at work: Program development in the Minlog system. In: Bibel, W., et al. (eds.) Automated Deduction, vol. II, pp. 41–71. Kluwer Academic Publishers (1998)
6. Bove, A., Capretta, V.: Modelling general recursion in type theory. Math. Struct. in Comp. Science 15, 671–708 (2005)
7. Bove, A., Dybjer, P., Sicard-Ramírez, A.: Embedding a Logical Theory of Constructions in Agda. In: PLPV '09. pp. 59–66 (2009)
8. Boyer, R.S., Kaufmann, M., Moore, J.S.: The Boyer-Moore theorem prover and its interactive enhancement. Computers & Mathematics with Applications 29(2), 27–62 (1995)
9. Dybjer, P.: Program verification in a Logical Theory of Constructions. In: Jouannaud, J.P. (ed.) Functional Programming Languages and Computer Architecture. LNCS, vol. 201, pp. 334–349 (1985)
10. Dybjer, P.: Comparing integrated and external logics of functional programs. Science of Computer Programming 14, 59–79 (1990)

11. Dybjer, P., Sander, H.P.: A functional programming approach to the specification and verification of concurrent systems. Formal Aspects of Computing 1, 303–319 (1989)
12. Gardner, P.: Representing Logics in Type Theory. Ph.D. thesis, University of Edinburgh, Department of Computer Science (1992)
13. Gordon, M.J., Milner, A.J., Wadsworth, C.P.: Edinburgh LCF. A Mechanised Logic of Computation, LNCS, vol. 78. Springer-Verlag (1979)
14. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. JACM 40(1), 143–184 (1993)
15. Harrison, W.L., Kieburtz, R.B.: The logic of demand in Haskell. Journal of Functional Programming 15(6), 837–891 (2005)
16. Krauss, A.: Partial and nested recursive function definitions in higher-order logic. Journal of Automated Reasoning 44(4), 303–336 (2010)
17. Lindblad, F., Benke, M.: A tool for automated theorem proving in Agda. In: Filliâtre, J.C., Paulin-Mohring, C., Werner, B. (eds.) Types for Proofs and Programs (TYPES 2004). LNCS, vol. 3839, pp. 154–169 (2006)
18. Martin-Löf, P.: Hauptsatz for the intuitionistic theory of iterated inductive definitions. In: Fenstad, J.E. (ed.) Proceedings of the Second Scandinavian Logic Symposium. pp. 179–216. North-Holland Publishing Company (1971)
19. Meng, J., Quigley, C., Paulson, L.C.: Automation for interactive proof: First prototype. Information and computation 204(10), 1575–1596 (2006)
20. de Mol, M., van Eekelen, M., Plasmeijer, R.: Theorem proving for functional programmers. Sparkle: A functional theorem prover. In: Arts, T., et al. (eds.) IFL 2001. LNCS, vol. 2312, pp. 55–71 (2002)
21. Park, D.: Finitess is mu-ineffable. Theoretical Computer Science 3, 173–181 (1976)
22. Paulson, L.C.: Isabelle. A Generic Theorem Prover, LNCS, vol. 828. Springer (1994), (With a contribution by T. Nipkow)
23. Peyton Jones, S. (ed.): Haskell 98 Language and Libraries: The Revised Report. Cambridge University Press (2003)
24. Plotkin, G.: LCF considered as a programming language. Theoretical Computer Science 5(3), 223–255 (1997)
25. Smith, J.: An interpretation of Martin-Löf's type theory in a type-free theory of propositions. The Journal of Symbolic Logic 49(3), 730–753 (1984)
26. Sutcliffe, G.: The TPTP problem library and associated infrastructure. The FOT and CNF parts, v.3.5.0. Journal of Automated Reasoning 43(4), 337–362 (2009)
27. Tammet, T., Smith, J.M.: Optimized encodings of fragments of type theory in first order logic. In: Berardi, S., et al. (eds.) TYPES '95. LNCS, vol. 1158, pp. 265–287 (1996)
28. The Agda development team: The Agda Wiki. Available at `http://wiki.portal.chalmers.se/agda` (2011)
29. The Nuprl development team: PRL Project. Available at `http://www.cs.cornell.edu/info/projects/nuprl/` (2011)
30. Turner, D.: An overview of Miranda. SIGPLAN Notices 21, 158–166 (1986)