Specifying Functional Programs with Intuitionistic First Order Logic

Marcin Benke

Institute of Informatics, University of Warsaw*

Abstract. We propose a method of specifying functional programs (in a subset of Haskell) using intuitionistic first order logic, that works well for inductive datatypes, higher-order functions and parametric polymorphism.

1 Introduction

Will we ever know the answer to all questions? Can one enter the same river twice? These and similar questions have been asked again and again since classical times and even long before that. Yet, ironically, the classical logic views the world as fixed and cognizable. A model of classical logic is a static structure with complete information about the relations between its elements. Such is the outlook that justifies the infamous *tertium non datur:* $p \vee \neg p$.

Software development is one of the disciplines, where one is recurringly, and often painfully reminded that such outlook is not only idealized, but often naïve: there is no one static world, but a multiverse of branching and merging worlds.; not only are we far from having all answers, they are rarely final even when we have them. Indeed there is a lot of bitter truth to the adage that the only constant is change.

Intuitionistic logic offers an attractive option for modelling a world of constant change and incomplete information. A Kripke model consists of a multitude of worlds, connected by an ordering relation. This relation may be time (earlier and later state of the world), but not necessarily so; nor need this order be linear; commit graph in a version control system gives a decent approximation, though we can and sometimes do consider models with infinite number of worlds as well.

One of the selling points of functional programming is its potential for easier and better specification and verification. While this potential is indisputable, the tools and methods to realise it are still lacking (see e.g. [1]).

Most approaches use either first order classical logic (e.g. [2, 6]) or higher-order logic (e.g. [4]). In this paper we propose a method of specifying Haskell programs using intuitionistic first order logic, that works well for inductive datatypes, higher-order functions and parametric polymorphism.

* This work has been supported by Polish NCN grant NCN 2012/07/B/ST6/01532.

2 The Logic

2.1 Core Logic

Our logic is essentially a variant of λP from [5] extended with constructs for existential quantifiers, alternative, conjunction and falsity. There are no separate constructs for implication or negation, as these can be easily encoded.

$$
\begin{array}{lll}
\Gamma & ::= & \{\} \mid \Gamma, (x : \phi) \mid \Gamma, (\alpha : \kappa) \\
\kappa & ::= & * \mid (\Pi x : \phi)\kappa \\
\phi & ::= & \alpha \mid (\forall x : \phi)\phi \mid \phi M \mid (\exists x : \phi)\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \bot \\
M & ::= & x \mid (\lambda x : \phi . M) \mid (M_1 M_2) \mid [M_1, M_2]_{\exists x:\phi.\phi} \mid \\
 & & \texttt{abstract}\ \langle x : \phi_1, y : \phi_2 \rangle = M_1\ \texttt{in}\ M_2 \mid \langle M_1, M_2 \rangle_{\phi_1 \wedge \phi_2} \mid \\
 & & \pi_1 M \mid \pi_2 M \mid \texttt{in}_{1,\phi_1 \vee \phi_2} M \mid \texttt{in}_{2,\phi_1 \vee \phi_2} M \mid \\
 & & \texttt{case}\ M_1\ \texttt{in}\ (\texttt{left}\ x : \phi_1 . M_2)(\texttt{right}\ y : \phi_2 . M_3) \\
 & & \varepsilon_\phi(M)
\end{array}
$$

2.2 Notation and Extensions

Additional connectives

$$a \rightarrow b \equiv \forall_{:} a.b$$

$$a \leftrightarrow b \equiv a \rightarrow b \wedge b \rightarrow a$$

$$\neg a \equiv a \rightarrow \bot$$

A Universe of values $V : \star$ is assumed. Quantifiers usually range over this universe, hence

$$\forall x.\phi \equiv \forall x : V.\phi$$

Axiom schemas of the form

```
schema name(P:kind) : formula
```

are to be understood as a finite set of formulas: one for every predicate symbol of the appropriate kind in the signature.

Proof sketches are rendered in a Mizar-like notation (cf. e.g. [3, 8, 7])

Haskell code is written using so called "Bird-tracks", e.g.

```
> id :: a -> a
> id x = x
```

Note The approach described in this document is "untyped" in the sense that we don't use Haskell type declarations, but derive our own types. Hence we might call our approach "owntyped" (there is also a strong connection with refinement types). On the other hand, we still use data type declaration as a source of useful information.

3 Datatypes

In this section we illustrate our method on some example Haskell datatypes and functions, starting with the simplest ones and progrsssing towards more complex ones.

3.1 Bool

```
> data Bool = False | True
```

We can characterize `Bool` by the following axiom

```
axiom defBool : ∀ x. Bool(x) ↔ x=False ∨ x=True
```

or by an axiom schema

```
schema elimBool(P):
  (P(False) ∧ P(True)) → ∀ x. Bool(x) → P(x)
```

Now consider the following definition

```
bnot False = True
bnot True = False
```

This definition can be characterized as follows

```
axiom defBnot : bnot False = True ∧ bnot True = False
```

Now let's prove that not takes *Bool* to *Bool* (in Mizar-like notation):

```
theorem typeBnot : ∀ x.Bool(x) → Bool(bnot x)
proof
  consider x st Bool(x)
  then x = False ∨ x = True by defBool
  thus thesis by cases
    suppose x = False
      then bnot x = True
      then thesis
    suppose x = True
      then bnot x = False
      thus thesis
end
```

An alternative proof of `typeBnot`, using `elimBool`

```
theorem typeBnot : ∀ x.Bool(x) → Bool(bnot x)
proof
  consider x st Bool(x)
  Bool(True) ∧ Bool(False) by defBool
  then Bool(bnot False) ∧ Bool(bnot True) by defBnot
  let P(x) = Bool(bnot x)
  thus thesis by elimBool(P)
end
```

This seems like an overkill and can probably be proved automatically. However, note that our statement is substantially stronger than a simple type assertion: it also states that `bnot` terminates for all inputs. Now, what about a function that doesn't? Consider

```
bad True = True
bad False = bad False
```

In Haskell, `bad :: Bool -> Bool`, but a theorem like

```
∀ x.Bool(x) → Bool(bad x)
```

is not provable. On the other hand, we can prove

```
theorem notSoBad : ∀ x.(Bool(x) ∧ x /= False)
                  → Bool(bad x)
```

3.2 Nat

```
> data Nat where { Z :: Nat; S :: Nat → Nat }

env Z, S : V
axiom introNat : Nat(Z) ∧ ∀ n. Nat n → Nat (S n)
schema elimNat (P:V→⋆) =
  ( P Z
  & ∀ n. Nat n → P n → P(S n)
  ) → ∀ m. P m
```

Alternative (and equivalent?) elimination

```
schema elimNat (P:V→⋆) =
 ( P Z
 & (∀ n. P n → P (S n))
 ) → ∀ m. P m
```

Now we can define some functions

```
> plus Z x = x
> plus (S n) x = S(plus n x)

axiom plusDef : ∀ x.plus Z x = x
              ∧ ∀ n x.plus (S n) x = S(plus n x)
```

Some properties

```
theorem plusType : ∀ x y. Nat(x) → Nat(y) → Nat(plus x
    y)
proof
  ∀ y. plus Z y = y by plusDef
  then ∀ y.Nat(y) → Nat(plus Z y)
  ∀ n x. Nat(plus n x) → Nat(S (plus n x)) by introNat
  then ∀ n x. Nat(plus n x) → Nat(plus (S n) x) by
     plusDef
  thus thesis by elimNat(P) where
    P n = ∀ y.Nat(plus n y)
end
predicate PlusZ(n : V) = plus x Z = x
theorem plusZR : ∀ n. Nat(n) → plusZ(n)
proof
  plus Z Z = Z by plusDef
  ∀ n.plus n Z = n → S(plus n Z) = S n by equality
  ∀ n.plus (S n) Z = S(plus n Z) by plusDef
  then ∀ n.plus n Z = n → plus (S n) Z = S n
  thus thesis by elimNat(plusZ)
end
```

3.3 Lists

To avoid confusion, we write the list type as `List a` and the corresponding predicate as `List` rather than use the usual `[a]`. In practice this is just amatter of syntactic sugar.

```
> data List a = Nil | Cons a (List a)
```

Lists can be axiomatised as follows:

```
env List : V→⋆ → ⋆, Nil : V, Cons : V → V
schema introList(T:V→⋆)
   = List(T)([])
   & (T(x)&List(xs) → List(Cons x xs))
schema elimList(T,P:V→⋆)
   =  P(Nil)
   &  (∀ x xs. T(x) & P (xs) → P(Cons x xs))
   → ∀ xs. List(T)(xs) → P(xs)
```

Sample theorem for map

```
> id x = x
> map f Nil = Nil
> map f (Cons x xs) = Cons (f x) (map f xs)

axiom mapDef : map f Nil = Nil & ∀ f x xs...
theorem mapType(T,U: V→⋆) : (∀ x. T(x) → U(f x))
```

```
                            →  (∀ xs. List(T)(xs) → List(U)
                                (map f xs))
theorem mapId(T:V→ ⋆) : ∀ xs.map id xs = xs
proof
  let P(xs:V) = map id xs = xs
  have P(Nil) & ∀ x xs.P(xs) → P(Cons x xs) by mapDef
  thus thesis by elimList(P)
```

Consider a (slightly convoluted) example of a function summing a list:

```
sum :: List Nat -> Nat
sum Nil = Z
sum (Cons n ns) = case n of
  Z     -> sum (Cons n ns)
  (S m) -> S(sum (Cons m ns))
```

this can be characterized as follows:

```
axiom sumNil : sum Nil = Z
axiom sumCons
 : ∀ n ns.  (n = Z →  sum (Cons n ns) = sum ns)
  ∧  (∀ m. n = S m
           →  sum (Cons n ns) = S(sum (Cons n ns))
```

4 Polymorphic Functions

If types translate to predicates, then one might think quantification over types might requiring quantifying over predicates. But we may avoid this reading “for all types a and values x of type a” as simply “for all x (regardless of type)”.

```
const :: a -> b -> a
const x y = x
--# axiom forall x y. const x y = x
```

5 Conclusions and Future Work

We have proposed a method of specifying Haskell programs using intuitionistic first order logic, that works well for inductive datatypes, higher-order functions and parametric polymorphism. On the other hand, one big remaining challenge is handling also ad-hoc polymorphism, i.e. type classes. One idea we’ve toyed with went along the following lines (in a notation slightly different to what we have used so far):

```
class Functor f where
  fmap :: forall a b.(a->b) -> f a -> f b

-- fmap_id :: forall a.f a -> Prop
```

```
-- # require fmap_id = forall x. fmap id x === x

instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just f x

-- # axiom Functor_im(Nothing)
-- # axiom forall x.Functor_im(Just x)

-- # conjecture forall i. Functor_im(i) -> fmap id i = i
```

This is not yet completely satisactory and needs more work.

References

1. Baranowski, W.: Automated verification tools for Haskell programs. Master's thesis, University of Warsaw (2014)
2. Claessen, K., Johansson, M., Rosén, D., Smallbone, N.: Hipspec: Automating inductive proofs of program properties. In: In Workshop on Automated Theory eXploration: ATX 2012 (2012)
3. Grabowski, A., Kornilowicz, A., Naumowicz, A.: Mizar in a nutshell. Journal of Formalized Reasoning 3(2) (2010), `http://jfr.unibo.it/article/view/1980`
4. Sonnex, W., Drossopoulou, S., Eisenbach, S.: Zeno: An automated prover for properties of recursive data structures. In: Flanagan, C., König, B. (eds.) Tools and Algorithms for the Construction and Analysis of Systems, Lecture Notes in Computer Science, vol. 7214, pp. 407–421. Springer Berlin Heidelberg (2012), `http://dx.doi.org/10.1007/978-3-642-28756-5_28`
5. Sørensen, M., Urzyczyn, P.: Lectures on the Curry-Howard Isomorphism, Studies in Logic and the Foundations of Mathematics, vol. 149. Elsevier (2006)
6. Vytiniotis, D., Jones, S.P., Rosén, D., Claessen, K.: Halo: Haskell to logic through denotational semantics. Acm Sigplan Notices 48:1, s. 431-442 (2013)
7. Wenzel, M., Wiedijk, F.: A comparison of Mizar and Isar. J. Autom. Reasoning 29(3-4), 389–411 (2002), `http://dx.doi.org/10.1023/A:1021935419355`
8. Wiedijk, F.: Formal proof sketches. In: Berardi, S., Coppo, M., Damiani, F. (eds.) Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers. Lecture Notes in Computer Science, vol. 3085, pp. 378–393. Springer (2003), `http://dx.doi.org/10.1007/978-3-540-24849-1_24`