Agda exercises
OPLSS

Peter Dybjer

Eugene, Oregon
June 2015

1 Booleans

1. Define implication of Booleans in Agda using pattern matching!

2. (a) Prove the law of the excluded middle for booleans:

   ```
   em : (b : Bool) -> So (b || (not b))
   ```

   where `||` is boolean disjunction.

   (b) Prove one of de Morgan's laws for booleans :

   ```
   dm : (a b : Bool)
     -> So (not (a && b) <==> (not a) || (not b))
   ```

3. SASL (an early lazy functional programming language) had a general tautology function. It is a higher order function which takes a boolean function (a predicate) of arbitrary arity as argument and checks whether it is a tautology, that is, whether it is true everywhere. With dependent types we can give it the following type

   ```
   taut : (n : N) -> Pred n -> Bool
   ```

   where `Pred` n is the type of n-place predicates:

   ```
   Pred zero     = Bool
   Pred (succ n) = Bool -> Pred n
   ```

   Your task is to program `taut` so that `taut` n f = `true` iff f is a tautology.

2 Natural numbers

4. (a) Define cut-off subtraction $\dot{-}$ in Agda, where $m \dot{-} n = 0$ if $m \leq n$!

   (b) Define the power function in Agda using pattern matching!

   (c) The recursion operator is a higher order function which takes a base case and a step case and returns a function defined by primitive recursion with that base case and step case. Its definition in Agda is

   ```
   natrec : {A : Set} -> A -> (Nat -> A -> A) -> Nat -> A
   natrec base step zero     = base
   natrec base step (succ n) = step n (rec base step n)
   ```

   Define the power function in terms of `natrec`!

5. (a) Define equality of natural numbers

   ```
   _==_ : Nat -> Nat -> Bool
   ```

   by pattern matching in both arguments.

   (b) Define the same function in terms of natrec! Note that this is harder, because natrec only does recursion in one argument at a time.

   (c) Write the Ackermann function in Agda! You can find its definition in wikipedia. There are actually several variants, but it suffices if you implement one of them. Compute the result of the Ackermann function for some arguments! Which is the largest number you can output?

3 Predicate logic

6. Implement proofs of the following propositions in Agda

   (a) $\neg\neg\neg P \rightarrow \neg P$

   (b) $\neg(P \vee Q) \leftrightarrow \neg P \& \neg Q$.

   (c) $\neg(\exists x \in D)P(x) \leftrightarrow (\forall x \in D)\neg P(x)$

   You should of course use the propositions as types interpretation of Curry and Howard! See for example chapter 4 in "Dependent types at work!!

7. Neither the law of excluded middle nor the law of double negation hold in *intuitionistic* logic.

   (a) However, excluded middle implies double negation:

   $$(A \vee \neg A) \rightarrow (\neg\neg A \rightarrow A)$$

   for all propositions A! Prove this in Agda (or on paper)!

(b) What happens if you try to prove the converse in intuitionistic logic (or Agda):

$$(\neg\neg A \to A) \to (A \vee \neg A)?$$

Discuss!

(c) Prove in Agda (or on paper) that if the law of double negation $\neg\neg X \to X$ holds for all propositions X, then the law of excluded middle $X \vee \neg X$ holds for all propositions X. (Note the difference to the formulation in (a)!)

(d) Also prove in Agda that if the law of excluded middle holds for all propositions X, then the law of double negation $\neg\neg X \to X$ holds for all propositions X. (Hint: this follows easily from (a).)

4 Logical Framework

8. Define some combinators using just the logical framework! (See e g wikipedia article.) Define dependently typed versions of them!

9. Define Church-Booleans. Which boolean operations can you define in a predicative framework?

10. Complete the definition of HOL (Church's simple theory of types)! (See e g the article on Church Type Theory in Stanford Encyclopedia of Philosophy (SEP).)

11. Encode more axioms of ZF! Prove some simple properties!

5 Lists, vectors, and trees

12. (a) Define the append function which concatenates two lists in Agda!

    (b) Define the append function on vectors, so that the type expresses that the dimension of the output vector is the sum of the dimensions of the input vectors. (Hint: beware that the type-checking algorithm normalizes (computes) the type, and is in this case therefore sensitive to the definition of addition. It matters whether addition is defined by recursion on the first or on the second argument.)

13. (a) Define the polymorphic reverse function in Agda! This can be done by quantifying over `a : Set`! This is a form of *explicit polymorphism*; since you explicitly need to quantify over all "sets" ("small types").

    (b) Define another polymorphic reverse function where `a : Set` is an *implicit argument*! With implicit arguments you can write functions in Agda with types looking much like the way you would write them in Haskell.

(c) In the lecture we defined the inductive family `Vec A n` of vectors of length `n` is defined. Define the reverse function on vectors, so that its type expresses that the length of the output vector is the same as the length of the input vectors. You can choose whether to define `Vec A n` either as a recursive family or as an inductive family as explained in "Dependent types at work" .

14. In section 3.2 in "Dependent Types at Work" the type `Fin n` of finite sets with `n` elements is define. Do the two exercises at the end of that section

    (a) Write a new lookup function `_!!_` so that it has the following type:

    ```
    _!!_ : {A : Set}{n : Nat}
      -> Vec A (succ n) -> Fin (succ n) -> A
    ```

    This will eliminate the empty vector case, but which other cases are needed?

    (b) Give an alternative definition of `Fin n` as a recursive family, that is, define it by induction on `n` using pattern matching!

15. An AVL-tree is a balanced ordered binary tree, where a binary tree is balanced provided the height difference between the left and right subtrees of any node is at most 1. Your task is to define the type of AVL-trees in Agda.

16. (a) Let `Term` of Boolean expressions (terms) be generated by the following grammar

    ```
    b ::= x | tt | ff | if b b b
    ```

    where x is one of denumerably many variables encoded by numbers. Implement Boolean expressions as a data type in Agda.

    (b) When we define the denotational semantics of Boolean expressions with variables, we do it relative to an *environment*, that is, a function which associates a value (a Boolean) to each variable:

    ```
    Env : Set
    Env = Nat -> Bool
    ```

    Hence the type of the interpretation function (the denotational semantics) is instead

    ```
    [[_]] : Term -> Env -> Bool
    ```

    Define this function in Agda!

    (c) Define a type `NandTerm : Set` in Agda of Boolean expressions built up only by nand and variables!

    (d) Define the interpretation function (the denotational semantics) for Boolean expressions built up from nand and variables.

    ```
    [[_]]' : NandTerm -> Env -> Bool
    ```

(e) Define a translation

```
tonand : Term -> NandTerm
```

which preserves the denotational semantics

(f) Prove in Agda that the the denotational semantics is preserved, that is, that

```
[[ tonand t ]]' env = [[ t ]] env
```

for all `t : Term` and all `env : Env`.

(g) Write the reverse translation

```
toif : NandTerm -> Term
```

and prove that this too preserves the denotational semantics.

17. Prove that the append function which concatenates two lists is associative.

18. Write an Agda program which terminates for all inputs but which is not accepted by Agda's termination checker!

19. (a) Implement the set of non-negative binary numbers in Agda or Haskell! You may implement them any way you like, but one possibility is to introduce a data type `BinaryNumber` with three constructors, using that each binary number is either zero, twice a binary number, or one more than twice a binary number.

(b) Write a function `bin2un` in Agda or Haskell which converts a binary number to a unary number!

(c) Write a function `un2bin` in Agda or Haskell which converts a unary number to a binary number! Hint: first write the successor function for binary numbers.

(d) Finally, prove that

```
bin2un (un2bin n) = n
```

in Agda. Why is not `un2bin (bin2un bs) = bs`?

6 Generalized inductive definitions

20. Each element of the type of Brouwer ordinals can be assigned a meaning as a von Neumann ordinal in classical set theory. Explain how! Define Brouwer ordinals corresponding to $\omega{+}1, \omega{+}2, \ldots \omega 2, \omega 2{+}1, \ldots, \omega^2, \ldots, \omega^3, \ldots, \omega^\omega, \omega^{\omega^\omega}, \ldots \epsilon_0$.

21. Implement natural numbers and Brouwer ordinals as suitable instances of the W-type! Can you derive the introduction and elimination-rules? Why or why not?

7 Identity and quotients

22. Prove that `So (m == n)` iff `m = n` where = is the inductively defined identity type given in the lecture.

23. (a) Define the set of integers `Int` in Agda and a function

    ```
    _==_ : Int -> Int -> Bool
    ```

    which tests for equality of integers. There are several ways of implementing the integers and it is up to you to choose one you like.

    (b) Define a function `nat2int` translating a natural number to an integer.

    (c) Define addition of integers `addInt`!

    (d) Prove that the addition of integers corresponds to addition of natural numbers is, that

    ```
    addInt (nat2int m) (nat2int n) == nat2int (m + n)
    ```

24. (a) What is a real numbers constructively? Look in the literature (e g on the web), choose a definition of the set `Real` of real numbers, and implement it in Agda!

    (b) Define a function

    ```
    eqReal : Real -> Real -> Set
    ```

    which defines when two of your constructive reals are equal.

    (c) Can you define a function

    ```
    decideEqReal : Real -> Real -> Bool
    ```

    which decides whether two of your constructive reals are equal? Why or why not?

8 Meaning explanations

25. Complete the semantics of System T in the file `Semantics.SystemT`!

    (a) First turn the postulates for substitution and instantation into proper definitions!

    (b) Define an inductive family of derivable judgments for System T! There should be one constructor for each inference rule.

    (c) Prove that these rules are valid in the model defined in the file `Semantics.SystemT`.

9 Records

26. Prove in Agda that any set with propositional identity is a setoid, by instantiating the record `Setoid`.

    (a) Write more axioms for the record of finite subsets? Can you find a "complete" set of axioms for the operations in question?

    (b) Add more operators for finite subsets, and axiomatize them? Cf Java Collections interface.

27. (a) Define an Agda record `FinMap` for finite maps, similar to the finite sets we defined in the lecture. Define some important operations on finite maps, and also add some axioms for these operations!

    (b) Implement your record by some simple data structure!