

# Draft Agda Document

[Agda-documentation team at AIST CVS]

February 24, 2005

## **Abstract**

This is a tutorial of *Agda*.

# 1 Basic operations for programming

Our aim in this section is to become familiar with the basic operations for developing `agda` code with `emacsagda`. By recreating the `agda` session described here, we are to gain an experience in constructing well-typed programs with those operations.

Explanations given here on semantics and syntax are minimal and rough; a more detailed description will appear in the planned reference manual.

## 1.1 Minimal set of commands

First we learn several basic commands. They let us communicate with `agda` in babble, that is, we can have `agda` recognize our codes, and have it compute some expressions.

An example `agda` code<sup>1</sup> is displayed in Figure 1. Before executions, let us check the code. The code consists of some blocks. Lines starting with the string “`--`” are comments, *except for* the first line. The first line is a directive to load another file. The second line is an open declaration, which declares to use names `List`, `nil`, ... defined in the package `OpList`. The rest of the code consists of four definitions of functions. We will see what each block means after checking Section 1.2.

The essential part of the code is the definition of `addElem`. For some type `X`, the function `(addElem X)` receives an element `x` in `X` and a list of lists

$$ys = [y_1, y_2, \dots, y_n],$$

and returns another list of lists

$$[(x : y_1), (x : y_2), \dots, (x : y_n)].$$

### 1.1.1 Starting/quitting `agda`

Now we are ready. To start up `agda` we can choose a way from the following:

- changing major mode to `agda-mode` by the command `M-x agda-mode`,
- opening a file with suffix “.agda” or “.alfa”.

in a Emacs buffer.

**Remark.** Suppose `agda` is invoked by the first way. Our code will not be recognized by `agda`, until we save it as a file explicitly. However if we deal with a code saved at least once — even if the code may be modified at that time —, then `agda` reads the code without any troubles.

When `agda` starts, we can see two windows and “Agda” menu in the Emacs menu bar. The larger window is called **main window**, in which you edit `agda` codes. The smaller window is called **sub window**, in which all the output from `agda` are displayed.

To quit `agda`, we should use the **quit** command `C-c C-q`. Owing to some reasons, if we want `agda` to restart without escaping from `agda-mode`, then we should invoke the **restart** command with `C-c C-x C-c`.

---

<sup>1</sup>We can get the code as a file somewhere in CVS’s LAN.

```

--#include "Op/List.alfa"

open OpList use List, nil, con, list, cat

-- function definition

addElem (X::Set):: X -> (List (List X)) -> (List (List X))
  = \(x::X) ->
    \(ys::List (List X)) ->
      case ys of
        (nil) -> con@_ (con@_ x nil@_) nil@_
        (con z zs) -> con@_ (con@_ x z) (addElem X x zs)

-- test data for evaluation
Color::Set = data Red | Blue | Green | Yellow | Black

aList::List (List Color)
  = con@_
    (con@_ Red@_ (con@_ Blue@_ nil@_))
    (con@_
      (con@_ Black@_ (con@_ Black@_ (con@_ Red@_ nil@_)))
      nil@_ )

foo::Color = {!addElem Color Blue@_ aList!}

```

Figure 1: Source code 1

## Summary

- To start up agda: Invoke `M-x agda-mode` or open a file with suffix “.agda” or “.alfa”
- To quite agda: `C-c C-q`
- To restart agda: `C-c C-x C-c`

### 1.1.2 Loading and computing

Of course before making agda load a code, it must be in a Emacs buffer. Namely, it must be typed directly, or opened from a file.

**Remark.** If we are typing directly, then we must not forget that agda is sensitive to indentations. Moreover, remember saving the code before going forward, since agda does not recognize unsaved codes.

Confirm that we are ready, and invoke the **chase-load** command by `C-c C-x RET`. By the command, agda reads the code in the current buffer, loads other files recursively (if needed), and checks the code.

Unless there is an error, some changes occur in Emacs. The string “Proof” appears in the status bar in the main window and the RHS of the last line changes from

```
{!addElem Color Blue@_ aList@_!}
```

to

```
{addElem Color Blue@_ aList@_}0
```

in a green box. In the sub window, the string

```
?0 :: Color
```

appears.

Now we are in “**Proof state**” where we can treat agda codes more interactively. The string “Proof” in the status bar indicates it. On the other hand, the state we were in before is called “**Text state**.” To go back to Text state, we should use the command `C-c ’`. All green boxes are converted to strings like as “{!...!}” and we can treat the code as a plain text.

Let us enter in Proof state, by invoking “chase-load” again. A green box in the code is called a **goal**. In general a goal is a place-holder to be filled with agda expressions. agda helps filling goals by inferring types, suggesting appropriate expressions, reducing expressions, etc. (We will practice those operations in Section 1.3). In the sub window, types of goals are displayed. A goal is indexed by a number presented at the tail of it. But indices change by commands, the order of invocations of commands, and so on. It does not matter if some indices are different from them in this tutorial.

Let us apply the function `addElem`. The argument `aList` is a list:

```
[ [ Red, Blue], [Black, Black, Red] ]
```

which is presented in agda

```
con@_ (con@_ Red@_ (con@_ Blue@_ nil@_))
      (con@_ (con@_ Black@_
              (con@_ Black@_ (con@_ Red@_ nil@_)))
        nil@_ )
```

Place the cursor on the goal and invoke the “**Compute to depth 100**” command by `C-c C-x +`. We will find a prompt in the minibuffer

```
expression: addElem Color Blue@_ aList
```

(In this example the goal is already filled, and thus the contents is displayed: c.f. Subsection 1.3) With typing RET, agda computes and displays the result in the sub window:

```
con@_
  (con@_ Blue@_ (con@_ Red@_ (con@_ Blue@_ nil@_)))
  (con@_
    (con@_ Blue@_ (con@_ Black@_ (con@_ Black@_ (con@_ Red@_ nil@_))))■
    (con@_ (con@_ Blue@_ nil@_ ) nil@_))
```

It is described in a familiar form:

```
[ [ Blue, Red, Blue ], [ Blue, Black, Black, Red], [ Blue ] ].
```

This is a walkthrough to use `agda`. Namely, we define a data type, a function and a constant, we apply the function to the constant, and we obtain expected results. Now we can modify or add other definitions (or write a new code from scratch) and play with `agda`.

## Exercises

## Summary

- There are Text-state and Proof-state in `agda`-mode.
- To switch to Proof-state, equivalently to make `agda` load your code, use `chase-load`: `C-c C-x RET`.
- Conversely, to switch to Text-state, use `C-c '`.
- To compute the reduced form of a goal, invoke `C-c C-x +` on the goal within Proof state

## 1.2 Brief explanations of the syntax

Before going forward to the more complicated example, syntax and expressions in `agda` are introduced. We can skip this section and come back repeatedly in reading the next section.

Like some functional programming languages, an `agda` code is a collection of definitions, and several directives. But there is important difference: `agda` is sensitive to the order of definitions. Names must not appear in such places that the definition follows.

**Comments** Comments are written in two ways.

1. A single line comment starts with “--” and ends at the tail of that line.
2. A block comment starts with “{-” and ends at the corresponding “-}”.

**Indentation** Like Haskell, `agda` is indentation-sensitive. Lines with the same indentations are regarded as in the same clause. The following two codes have the same meaning.

- With indentations

```
case x of
  (nil ) -> true@_
  (y:ys) -> false@_
```

- Without indentations

```
case x of { (nil ) -> true@_ ; (y:ys) -> false@_ }
```

**Include a file** Definitions in another file are included by the directive “`--#include`”. For example, to include definitions in the file “`aFile.agda`” put the line

```
--#include "aFile.agda"
```

at the head of an `agda` code. Although this line starts with “`--`”, it is not a comment. Files are searched in

- the current directory
- the include path, set in the variable `agda-include-path`.
- the file itself, if the absolute path is specified.

**Remark.** An error occurs if the `agda` code has no definitions and no `open` directives but “`--#include`” directives.

**Remark.** When there is something wrong in a included file, for example non-existence of it, an error in it, and so on, `agda` does not load the current buffer and does not specify that.

**Open declarations** Names defined in other packages are referred with a dot like as `aPackage.name`. The `open` statement allows an `agda` code to refer it to `name` simply. The syntax is

```
open aPackage use name1 :: type1, name2 :: type2, ...
```

where `name1`, `name2`, ... are defined in the package `aPackage`. Type annotations may be omitted.

For example, the following two codes are equivalent.

- With `open` statement

```
--#include "Op/List.alfa"
--#include "Op/Bool.alfa"

open OpBool use Bool, true
open OpList use List, con, nil

foo::List Bool = con Bool
                  true
                  (nil Bool)
```

- Without `open` statement

```
--#include "Op/List.alfa"
--#include "Op/Bool.alfa"

foo::OpList.List OpBool.Bool
    = OpList.con OpBool.Bool
      OpBool.true
      (OpList.nil OpBool.Bool)
```

We can assign a local name to a name in other packages. In the following line,

```
open aPackage use lodef=padef
```

the name `padef` defined in the package `aPackage` is referred to `lodef` in the current buffer.

**Definitions** A definition (except for a Package) in an `agda` code has a form

$$name :: type = body$$

There is no differences to define names with different kinds, for example functions, data types, or types.

- A constant function:

$$aConst :: Bool = true@_$$

- A function:

$$\begin{aligned} aFunc &:: Bool \rightarrow Nat \\ &= \lambda(x :: Bool) \rightarrow \\ &\quad \text{case } x \text{ of} \\ &\quad \quad (true) \rightarrow zer@_ \\ &\quad \quad (false) \rightarrow suc@_ zer@_ \end{aligned}$$

- A data type definition:

$$aData :: Set = \text{data Red | Blue}$$

- A type definition:

$$aType :: Set = List Nat$$

In general an expression depends on other expressions. A definition of a name depending on other expressions is as follows:

$$name(var_1 :: type_1)(var_2 :: type_2) \cdots (var_n :: type_n) :: type = body.$$

This is equivalent to

$$\begin{aligned} name &:: (var_1 :: type_1) \rightarrow (var_2 :: type_2) \rightarrow \cdots (var_n :: type_n) \rightarrow type = \\ &\quad \lambda(var_1 :: type_1) \rightarrow \lambda(var_2 :: type_2) \rightarrow \cdots \lambda(var_n :: type_n) \rightarrow body \end{aligned}$$

**Types and type constructors** Any legal expression in an `agda` code must be of a legal type. Here we give a brief summary of types in `agda`.

The type formation rules in `agda` are as follows:

- `Set` is a type.
- An object of `Set` is a type, which particularly is a ‘set’, also known as a ‘small type.’
- $(x :: A) \rightarrow B$  is a type, called “dependent function type”, provided that  $A$  is a type and that  $B$  (which may contain free occurrences of  $x$ ) is a type under the assumption  $x :: A$ .

**Remark.** When  $B$  is a type which does not depend on  $x$ , then the type  $A \rightarrow B$  is defined as an abbreviation of  $(x :: A) \rightarrow B$ . It is a function type.

Small types are closed under dependent function type formation, meaning that when  $A$  and  $B$  are sets,  $(x :: A) \rightarrow B$  is also a set<sup>2</sup>.

<sup>2</sup>In general types are stratified by their “size”: type expressions must have a type, but saying all types have the type `Set`, including `Set` itself, would lead to paradoxes.

- Listing up objects defines a type, which is an object of **Set** (See Paragraph **Data constructors**).
- A signature, which is a collection of names with types, defines a type. (C.f. Paragraph **Struct**.) The syntax is

```
sig
  name1 :: type1
  name2 :: type2
  ...
```

**Data constructors** Let us start the simplest definition. The statement

```
labelid :: Set = data conid1 | conid2 | conid3
```

defines a type *labelid* and objects of it. In precise, this statement means that “the type of an expression *expr* is *labelid* if and only if *expr* is definition-ally equal to one of *conid<sub>1</sub>*, *conid<sub>2</sub>* or *conid<sub>3</sub>*.” Thus two constructors which have the same name but have different types are allowed.

A constructor may take some arguments, and thus its type may depend on some expressions. Hence a data type definition has the following form in general:

```
labelid (var1 :: type1) (var2 :: type2) ...
= data const1 | const2 | const3 | ...
```

where *const<sub>i</sub>* is

```
conidi (argi,1 :: typei,1) (argi,2 :: typei,2) ...
```

The type *type<sub>2</sub>* may depend on *x :: type<sub>1</sub>*, the argument *type<sub>i,2</sub>* may depend on *y :: type<sub>i,1</sub>*, and so on.

For example, natural numbers are defined by a nullary constructor and a unary constructor:

```
Nat :: Set = data Z | S (n :: Nat)
```

**agda** allows *inductive families of types*. They are types dependent on arguments of their constructors. For example, let *A* be a type and consider the type of lists of *A* with length *n*, denoted by

```
Vec(A :: Set)(n :: Nat)
```

Its constructors are

- **Zero** for  $n = 0$ ,
- **cons** ( $a :: A$ ) ( $v :: \text{Vec } A \ m$ ) for  $n = m + 1$ .

Although we cannot define  $\text{Vec}(A :: \text{Set})(n :: \text{Nat})$  by one **data** statement, we can do it by **idata** statement:

```
Vec (A :: Set) :: Nat -> Set
= idata Zero :: _ zer |
      cons (n :: Nat) (a :: A) (v :: Vec A n) :: _ suc n
```

Namely, strings after “`::` `_`” works as indices:

```
Zero@_ :: Vec A zer
cons (a::A) (v::Vec A n) :: Vec A (suc n)
```

In general, an inductive family of types is defined by

```
name pardecl :: typeP = i data icondef1 | icondef2 | ...
```

In the statement, *pardecl* is a declaration of parameters, *typeP* is a product type dependent on parameters, and *icondef<sub>i</sub>* is

```
iconid argdecl :: _ index
```

where *argdecl* is an argument declaration which is also dependent on parameters, and *index* must be an operand of *typeP*.

**Remark.** (1) Data constructors are referred by the names with suffix “`@_`”. except for some cases.

(2) An `(i)data` statement can be placed in which an RHS expression can be placed. However it is not recommended that an `(i)data` statement is placed except for the right of “`=`.”

**Function applications** It is expressed by juxtaposition. Let  $f :: A \rightarrow B$  and  $expr :: A$  be expressions in `agda` where  $B$  does not depend on  $x :: A$ . Then the juxtaposition  $f \text{ expr}$  is an object of  $B$ . Moreover, let  $f :: (a :: A) \rightarrow B$  and  $expr :: A$  be expressions in `agda`. Then juxtaposition  $f \text{ expr}$  is an object of  $B \text{ expr}$ .

**Abstractions** The expression

```
\(varname :: typename) -> expression
```

means a function, or equivalently means an abstraction: “for a bound variable *varname* of type *typename*, *expression* is an `agda` term.”

**Case expressions** A case expression is written as

```
case expr of
  (case1) -> expression1
  (case2) -> expression2
  ...
```

Each branch should correspond to a constructor, thus *case<sub>i</sub>* is *constid arg<sub>1</sub> arg<sub>2</sub> ...* whose type is as same as *expr*. An error occurs otherwise. ■

There are some remarks:

- Branching must be total unlike Haskell, that is *expr* must match one of the branches.
- As *expr* lets `agda` infer the type of *case<sub>i</sub>*, the string “`@_`” following constructors is not necessary.
- A case expression may not work as expected, if *expr* is so complicated that `agda` cannot infer the type. A solution for this problem is to (locally) define a small function that just does case on its argument variable and give *expr* as an actual argument to it.

**Let expression** Let expressions make it possible to use local variables.

```
let name1 :: type1 = expr1
    name2 :: type2 = expr2
    ...
in
  body
```

The name  $name_i$  has effects only in the `in` clause.

**Struct** A `struct` expression defines a term of a signature type:

```
struct name1 :: type1 = value1
      name2 :: type2 = value2
      ...
```

Needless to say a signature type

```
sig
  {\it name}_1\tttdc{\it type}_1
  {\it name}_2\tttdc{\it type}_2
  ...
```

must be defined in the earlier part of the code.

To access a member in an object of a signature type, a dot expression  $typename.memname$  is used.

```
Tuple::Set = sig fst::Bool
              snd::Bool

aPair::Tuple = struct fst::Bool = true@_
                  snd::Bool = false@_

foo::Bool = aPair.fst
```

### 1.3 Editing assistant

The next lesson to communicate with `agda` is to learn operations to make codes interactively.

The complete code we make here is displayed in Figure 2. It includes the code we have already typed in the previous section. The most essential block in the code is the definition of `subList`. It is a function which receives a list and returns the list of all sublists of it.

#### 1.3.1 Editing code

Let us start with the definition of `subList`. Confirm that we have typed the earlier part (Line 1 – Line 16) by hand (or cut-and-paste). We can see the following lines in a Emacs buffer.

---

```

--#include "Op/List.alfa"

open OpList use List, nil, con, list, cat

-- data constructor
Color::Set = data Red | Blue | Green | Yellow | BLACK

-- function definitions

addElem (X::Set):: X -> (List (List X)) -> (List (List X))
  = \ (x::X) ->
    \ (ys::List (List X)) ->
      case ys of
        (nil) -> con@_ (con@_ x nil@_) nil@_
        (con z zs) -> con@_ (con@_ x z) (addElem X x zs)

subList::(List Color)->(List (List Color))
  = \ (ys::List Color)->
    case ys of
      (nil) ->
        nil@_
      (con x xs)->
        let zs::List (List Color)
          = subList xs
        in cat (List Color)
            zs
            (addElem Color
              x
              zs)

-- temporary definitions for evaluations

dummy::List Color =con@_ Red@_ (con@_ Green@_ (con@_ Blue@_ nil@_))
foo:: List (List Color) = {!!}

```

Figure 2: Source code

```

--#include "Op/List.alfa"

open OpList use List, nil, con, list, cat

-- data constructor
Color::Set = data Red | Blue | Green | Yellow | BLACK

-- function definitions

addElem (X::Set):: X -> (List (List X)) -> (List (List X))
  = \(x::X) ->
    \(ys::List (List X)) ->
      case ys of
        (nil) -> con@_ (con@_ x nil@_) nil@_
        (con z zs) -> con@_ (con@_ x z) (addElem X x zs)

```

---

**Refining a goal.** Now we are ready. Let us type the following two lines, appending to lines we have input, in which the metavariables ‘?’ appear in the type and the body.

```

subList:: ?
  = ?

```

**Remark.** Whitespaces (including a carriage return, or a tab) must be put in the place adjacent to ?, or `agda` does not recognize the metavariable.

Whenever we want to cancel the last command, we can use the **undo** command invoked by `C-c C-u`.

Next invoke the **chase-load** command. Check the two copies of the symbol ‘?’ change to goals `{ }0` and `{ }1`. We will refine goals step by step as follows.

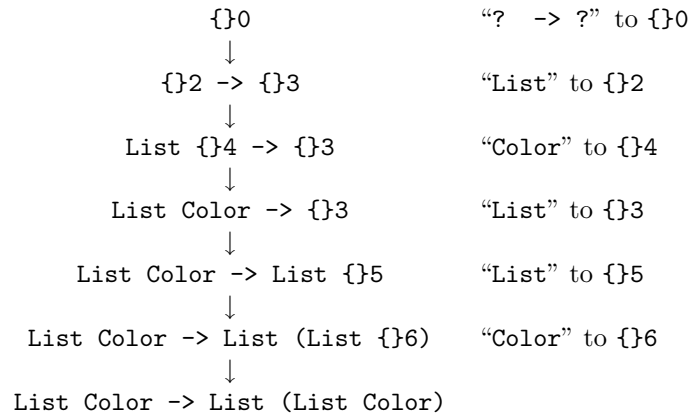
The goal `{ }0` is the type declaration of `subList`. Since its type is a function type, first we should refine the goal to  $type_1 \rightarrow type_2$ . Fill `{ }0` with the string “? -> ?” and try invoking the **refine** command `C-c C-r` on the goal. What occurs? The goal `{ }0` is “refined” to the expression “`{ }2 -> { }3`.” Next, fill the goal `{ }2` with the string “List” and invoke refine command again. The result is

```

List { }4 -> { }3

```

which is more refined expression. Similarly, the goal `{ }0` is completely refined as follows:



**Templates, solving** Next we must fill the goal { }1, which is the function body. Fill the goal with the string “**ys**”, which is the argument to `subList` and invoke **abstraction** command by `C-c C-a` on the goal. Then the goal { }1 changes to

```
= \ (ys :: { }7) ->
   { }8
```

Namely, this command gives a template of an abstraction expression. We can easily see that the goal { }7 must be filled with “`List Color`,” since `subList` receives an argument of that type. `agda` can also see that, and it knows what to do. Invoke **solve** command with `C-c =` on the goal { }7.

```
subList :: (List Color) -> (List (List Color))
= \ (ys :: List Color) ->
   { }8
```

Like the abstraction command, **case** command (`C-c C-c`) and **let** command (`C-c C-l`) give templates respectively.

```

subList::(List Color)->(List (List Color))
  = \(ys::List Color) ->
    {}8

```

↓

```

subList::(List Color)->(List (List Color))
  = \(ys::List Color) ->
    case ys of
      (nil      )->
        {}9
      (con x xs)->
        {}10

```

↓

```

subList::(List Color)->(List (List Color))
  = \(ys::List Color) ->
    case ys of
      (nil      )->
        nil@_
      (con x xs)->
        {}10

```

↓

```

subList::(List Color)->(List (List Color))
  = \(ys::List Color) ->
    case ys of
      (nil      )->
        nil@_
      (con x xs)->
        let zs::{}11
          = {}12
        in {}13

```

To fill the rest of goals is an easy exercise.

**Remark.** In Proof state, we can edit the code other than goals. Especially, variables given by `agda` commands can be renamed. Remember that we should invoke the chase-load command again after modifying a code.

**Evaluations** When the code is typed in completely, try applying `subList` like the previous section.

By invoking `chase-load`, only one goal appears in the last line. Place the cursor on the goal, and invoke `compute to depth 100` by `C-c *`. Then the prompt

```
expression:
```

appears in the minibuffer. We should input

```
expression: subList dummy
```

with RET, and the result is obtained:

```
con@_
  (con@_ Blue@_ nil@_)
  (con@_
    (con@_ Green@_ (con@_ Blue@_ nil@_))
    (con@_
      (con@_ Green@_ nil@_)
      (con@_
        (con@_ Red@_ (con@_ Blue@_ nil@_))
        (con@_
          (con@_ Red@_ (con@_ Green@_ (con@_ Blue@_ nil@_)))■
          (con@_
            (con@_ Red@_ (con@_ Green@_ nil@_))
            (con@_ (con@_ Red@_ nil@_) nil@_))))))
```

In invoking the compute command, the expression is not restricted. We can compute even an expression whose type differs from the goal's. Let us see that. Again invoke the “compute to depth 100” command on the goal, and give to the prompt the string

```
cat (List Color) (con@_ Red@_ nil@_) (con@_ Blue@_ nil@_)
```

whose type is List Color. agda returns the result without any errors.

## Summary

- Making a code step by step, by refining goals with `C-c C-r`.
- Templates for `let`, `case`, and abstractions.
- Computing various expressions at a goal.

**Exercises** 1. Make the code figured in Figure 3. Some lines and words (names, ‘=’, “package”, etc.) must be input directly, but many parts in the code are given by agda commands.

2. With the package that you made in the previous exercise, make an agda code to apply K to S. Check the value of the evaluation K S.

## 1.4 List of commands

All agda commands can be invoked by key operations, or by selecting items in menus. The commands which is effective in the whole of the code are found in Agda menu in the menu bar. On the other hand, the commands for goals are found in the popup menu by right-clicking on a goal. Most of item in the goal menu depend on the context.

Commands are classified to four categories roughly.

**Necessary** commands you must know.

**Important** commands used very often.

```

package ULambda where
T::Set = data L (f::T -> T)
app::T -> T -> T
  = \(f::T) -> \(g::T) ->
      case f of
        (L f) -> f g
K::T = L@_ (\(x::T) -> L@_ (\(y::T) -> x))
S::T = L@_ (\(z::T) ->
  L@_ (\(y::T) ->
    L@_ (\(x::T) ->
      app (app x z) (app y z))))

```

Figure 3: Exercise 1

**Often** commands which help you use `agda` effectively. You can do without them.

**Not recommended.** Not sophisticated, or complicated commands. They may be changed or removed in the future.

#### 1.4.1 Agda menu

##### Restart

key: C-c C-x C-c

category: *often*

(Re-)initializes the type-checker.

##### Quit

key: C-c C-q

category: *necessary*

Quits and cleans up after agda. If you do not want Emacs to warn in quitting Emacs, then you should invoke this command every time.

##### Goto error

key: C-c `

category: *important*

Jumps to the line the first error occurs.

##### Load

key: C-c C-x C-b

category: *often*

Reads and type-checks the current buffer.

##### Chase-load

key: C-c C-x RET

category: *necessary*

Reads and type-checks the current buffer and included files.

##### Chase-import

key: C-c C-x TAB

category: *not recommended*

Reads and type-checks quickly the current buffer and included files.

### **Solve**

key: C-c =

category: *often*

Solves constraints that have unique solutions (Ref. Show constraint).

### **Solve Constraint**

key: C-c C-x =

category: *not recommended*

Solves constraints that have unique solutions with some tactics. In invoking this command, tactics and a constraint must be chosen. Tactics should be 0 in current version. Constraints are indexed by numbers (Ref. Show constraint).

### **Unfold constraint**

key: C-c C-x C-e

category: *not recommended*

Shows the unfolded expression in a constraint. In invoking this command, a constraint and which side of it is unfolded must be chosen (Ref. Show constraint).

### **Show constraints**

key: C-c C-e

category: *often*

Shows all constraints in the code. A constraint is an equation of two goals or of a goal and an expression. Each constraint is indexed by a number.

### **Suggest**

key: C-c C-x C-s

category: *not recommended*

Suggests suitable expressions.

### **Show goals**

key: C-c C-x C-a

category: *often*

Shows all goals in the current buffer.

### **Next goal**

key: C-c C-f

category: *often*

Moves the cursor to the next goal, if any.

### **Previous goal**

key: C-c C-b

category: *often*

Moves the cursor to the previous goal.

### **Undo**

key: C-c C-u

category: *important*

Cancels the last operation.

### **Text state**

key: C-c '

category: *necessary*

Resets agda to the state before the current buffer is loaded.

**Check termination**

key: C-c C-x C-t

category: *often*

Runs termination check on the current buffer.

**Submitting bug report**

key:

category:

(not implemented)

**1.4.2 Goal commands.**

**Give**

key: C-c C-g

category: *often*

Gives to the goal the expression in it.

**Intro**

key: C-c TAB

category: *often*

Introduces abstraction or struct in a goal. Opposite to the Abstraction command, this command give a template with a variable name.

**Refine**

key: C-c C-r

category: *important*

Refines the expression in the goal.

**Refine(exact)**

key: C-c C-s

category: *not recommended*

Saturates and gives to the goal the expression in it.

**Refine(projection)**

key: C-c C-p

category: *often*

Refines the goal with given package and projection. For example, a function `cat` is defined in the package `OpList`. When a goal is filled with “`OpList cat`”, the Refine (projection) command accepts it and refine the goal but the Refine command fails.

**Case**

key: C-c C-c

category: *often*

Makes a template of a case expression with a given variable name.

**Let**

key: C-c C-l

category: *often*

Makes a template of a let expression with a given variable name.

### Abstraction

key: C-c C-a

category: *often*

Makes a template of a function expression with a given variable name.

### Goal type

key: C-c C-t

category: *often*

Shows the type of the goal.

### Goal type(unfolded)

key: C-c C-x C-r

category: *often*

Shows the reduced type of the goal.

### Context

key: C-c |

category: *important*

Shows context (names already defined) of the goal.

### Infer type

key: C-c :

category: *important*

Prompts an expression and infers the type, under the context.

### Infer type(unfolded)

key: C-c C-x :

category: *often*

Prompts an expression and infers the reduced type, under the context.

### Unfold one

key: C-c +

category: *not recommended*

Prompts an expression and unfolds it one step.

### Continue one step

key: C-c c

category: *not recommended*

Continues one more step the last result of Continue/Unfold/Compute.

### Continue several steps

key: C-c C-x c

category: *not recommended*

Unfolds several more step the last result of Continue/Unfold/Compute.

### Compute WHNF

key: C-c \*

category: *not recommended*

Prompts an expression and computes its weak head normal form.

### Compute WHNF strict

key: C-c #

category: *not recommended*

Prompts an expression and computes its weak head normal form strictly.

**Compute to depth 100**

key: C-c C-x +

category: *important*

Prompts an expression and computes it to depth 100.

**Compute to depth**

key: C-c C-x 2 +

category: *not recommended*

Prompts an expression and computes it to given depth.

## 2 Proving in basic logics.

In this section, we shall discuss proving in logics. We shall give basic type theory from the subsections 2.1 to 2.6, and the examples of proofs in 2.7. We suggest that those who know logic should start from the subsection 2.7.

### 2.1 Propositions as sets.

In this section, we shall prove theorems in basic logics using `agda`. The `agda` is an implementation of  $\lambda$ -calculus in Martin L of Type theory. Hence it can provide, via Curry-Howard correspondence, proofs in higher-order Intuitionistic logic. An important aspect of Intuitionistic logic is its constructivity, which is formulated mathematically as the **disjunction property** and **existence property**. The disjunction property says that if a formula  $A \vee B$  is provable, then either  $A$  or  $B$  is provable. Similarly, The existence property says that if a formula  $\exists x A$  is provable, then there exists a term  $t$  such that  $A[t/x]$  is provable.

We shall work within the paradigm called ‘‘Propositions as sets’’. Refer to [1] for thorough discussions on it.

The `SET` package in `Hedberg/SET.alfa` provides us this paradigm: Any proposition is not syntactically defined, but is semantically defined as the set of its proofs constructed from function spaces, products and coproducts, and so on. This interpretation of logic in type theory is called **shallow embedding**.

Among many basic definitions, a part of the `SET` package in `Hedberg/SET.alfa` is devoted to the logical constants.

```
----- ■
-- Propositions.
----- ■

Imply (X::Set)(Y::Set) :: Set
  = X -> Y
Absurd :: Set
  = Zero
Taut :: Set
  = Unit
Not (X::Set) :: Set
  = X -> Absurd
Exist (X::Set)(P::Pred X) :: Set
  = Sum X P
Forall (X::Set)(P::Pred X) :: Set
  = (x::X) -> P x
And (X::Set)(Y::Set) :: Set
  = Times X Y
Iff (X::Set)(Y::Set) :: Set
  = And (Imply X Y) (Imply Y X)
Or (X::Set)(Y::Set) :: Set
  = Plus X Y
```

In the list above, any variable has either type `Set` or `Pred X`. Due to our ‘‘propositions as sets’’ perspective, `Set` is also the type for Propositions: We shall use the type `Prop` of propositions and `Set` interchangeably from now on. The

constant `Pred` is the type of unary predicates, and is defined as the following function type in the SET package:

```
Pred (X::Set) :: Type
  = X -> Set
```

The set `X` here is used as the domain of individuals in semantics for the first-order predicate logic. The non-emptiness of domain `X` is often an assumption, though is not a primary one. This is established by way of a package definition, which we shall discuss in the next section.

The expression of a formula  $P(a)$  for a unary predicate  $P$  and a term  $a$  is obtained in `agda` as follows. Let `A`, `a` and `P` be expressions of types `Set`, `A` and `Pred A`, respectively. Then, by definition, `P a` is an expression of type `Set`, the proposition type.

Here is a correspondence between the above logical constants in the SET package and logical connectives:

<code>Imply</code>	$\supset$ (Implication)
<code>Absurd</code>	$\perp$ (Falsity)
<code>Taut</code>	$\top$ (Truth)
<code>Not</code>	$\neg$ (Negation)
<code>Exist</code>	$\exists$ (Existential quantifier)
<code>Forall</code>	$\forall$ (Universal quantifier)
<code>And</code>	$\wedge$ (Conjunction)
<code>Iff</code>	$\equiv$ (Equivalence)
<code>Or</code>	$\vee$ (disjunction)

Let  $A$  be a proposition. In “Propositions as Sets” perspective, statements “the proposition  $A$  holds / is true”, “a proof of  $A$  can be constructed” and “the set  $A$  is inhabited” are all equivalent to one another. Let `A` be the corresponding expression to  $A$ : Then an `agda` proof or **proof object** has type `A`. In the following subsections, we shall analyze each definition more closely.

## 2.2 Constants.

Logical constants in this category are `Absurd` and `Taut`:

### 2.2.1 Absurdity.

The Falsity (Absurdity) is a logical constant that is always false. Hence it is defined to be the empty set, expressed as `Zero` in the SET package.

```
Absurd :: Set = Zero
```

### 2.2.2 Truth.

The Truth is a logical constant that is always true: Hence it is defined to be a non-empty set. In the SET package, it is a singleton set `Unit`.

```
Taut :: Set = Unit
```

## 2.3 Function types.

The logical constants in this category are `Imply`, `Negation` and `Forall`:

### 2.3.1 Universal Quantifier.

The term `Forall X P` is defined by function type  $(x :: X) \rightarrow P\ x$ .

```
Forall (X :: Set) (P :: Pred X) :: Set
= (x :: X) -> P x
```

We shall discuss a slightly more general type  $(x :: A) \rightarrow B$ . Let  $A$  be a set and  $B$  (or we may write  $B(x)$ ) a predicate possibly containing a free variable  $x$ . Let  $A$  and  $B$  be the corresponding `agda`-expressions for  $A$  and  $B$  of type `Set` and `Pred A`, respectively.

Then the function type  $(x :: A) \rightarrow B$  is an `agda`-expression of the **dependent product**  $\prod_{x \in A} B(x)$ : The dependent product is the type of functions, which, for a given  $x \in A$ , returns an element in  $B(x)$ .

Let  $A$  be a set with `agda` expression  $A$  of type `Set`, and  $P$  be a predicate. By using the expression  $P :: \text{Pred } A$  for the  $P$  in `agda`,  $\forall x :: A. P(x)$  is expressed as `Forall X P`. Since `Forall X P` is defined to be  $(x :: X) \rightarrow P\ x$ , a proof of  $\forall x :: A. P(x)$  is a function defined on  $A$  which assigns any object  $a$  in  $A$  to a proof of  $P(a)$ .

Let  $b :: B$ : A canonical object of function type  $(x :: A) \rightarrow B$  is a **lambda expression** of the form  $\lambda(x :: A) \rightarrow b :: (x :: A) \rightarrow B$ .

This is based on the Curry-Howard correspondence, and we discuss it in the next subsection.

### 2.3.2 Implication.

The logical constant `ImPLY` is an `agda`-expression for logical connective  $\supset$  (Implication): It is a binary operator defined as follows.

```
ImPLY (X :: Set) (Y :: Set) :: Set
= X -> Y
```

`ImPLY X Y` is defined as (non-dependent) function type  $X \rightarrow Y$ , which is a shorthand of  $(x :: X) \rightarrow Y$  when  $Y$  is non-dependent of  $x$  of  $X$ . This is based on the Curry-Howard correspondence: The proofs of proposition  $X \supset Y$  in intuitionistic natural deduction (NJ) are identified with typed  $\lambda$ -terms of type  $X \supset Y$ . In other words, a proof of the implication is a function that constructs a proof of  $Y$  from that of  $X$ .

Let  $A$  and  $B$  be sets with `agda`-expressions  $A$  and  $B$  of type `Set`, respectively. Let  $x$  be a variable over  $A$  and  $b$  be an element of  $B$ , with `agda`-expressions  $x :: A$  and  $b :: B$ . Under the Curry-Howard correspondence, the term  $\lambda x : A. b$ , obtained by  $\lambda$ -abstraction, has type  $A \supset B$ . The corresponding `agda`-expression is  $\lambda(x :: A) \rightarrow b :: A \rightarrow B$ .

### 2.3.3 Negation.

In Intuitionistic logic,  $\neg A$  is defined as  $A \supset \perp$ . In the `SET` package, the corresponding definition in `Agda` is:

```
Not (X :: Set) :: Set = X -> Absurd
```

## 2.4 Sum types.

The sum type `Sum` is defined by **signature type**:

```
Sum (X::Set)(Y::X -> Set) :: Set
  = sig{fst :: X;
        snd :: Y fst;}
```

Let  $Y(x)$  be a type depending on  $x$ . The expression `Sum X Y` represents the **dependent coproduct**  $\coprod_{x \in X} Y(x)$ . This is the (tagged) disjoint union of the elements in  $Y(x)$  for all  $x$  in  $X$ . Thus to construct an object of this type is to construct an object of type  $Y(x)$  for some  $x$  in  $X$ , which, in turn, is to prove the proposition that “there exists  $x \in X$ ,  $Y(x)$  (holds).”

The canonical objects of this type are defined using **structures** with labels `fst` and `snd`.

```
dep_pair (X::Set)(Y::X -> Set)(x::X)(y::Y x) :: Sum X Y
  = struct {
    fst = x;
    snd = y;}
```

Logical constants `Exist` and `And` in the `SET` package are defined by means of the sum type.

### 2.4.1 Existential Quantifier.

The logical constant `Exist` is an `agda`-expression for logical connective  $\exists$ : It is a binary operator defined for variables `X :: Set` and `P :: Pred X`. The term `Exist X P` is defined as `Sum X P` itself in the `SET` package:

```
Exist (X::Set)(P::Pred X) :: Set
  = Sum X P
```

The canonical proof object of type `Exist A P` is represented as a dependent pair `dep_pair A P a b`, or `struct {fst = a; snd = b;}`, consisting of a of type `A` and `b` of type `P a`. Conversely, if `ap` is a proof object of type `Exist A P`, then `ap.fst` and `ap.snd` are objects of type `A` and `P e`, respectively. Hence, `Exist A P` has a proof object if and only if there are objects `a` of type `A` and of type `P a`. This is the *existence property* in Intuitionistic logic.

### 2.4.2 Conjunction.

The logical constant `And` is an `agda`-expression for logical connective  $\wedge$  (Conjunction): It is a binary operator defined for variables `X` and `Y` of type `Set`.

```
And (X::Set)(Y::Set) :: Set
  = Times X Y
```

The binary operator `Times` is defined by

```
Times (X::Set)(Y::Set) :: Set
  = Sum X (\(x::X) -> Y)
```

Let  $Y :: \text{Set}$ : Then  $\lambda(x :: X) \rightarrow Y$  has type  $X \rightarrow \text{Set}$  and  $\text{Sum } X (\lambda(x :: X) \rightarrow Y)$  represents the coproduct  $\coprod_{x \in X} Y$ . Let  $\text{ab}$  be an object of this type. Then  $\text{ab.snd}$  has type  $(\lambda(x :: X) \rightarrow Y) \text{fst}$ , which  $\beta$  reduces to  $Y$  itself. Thus the coproduct  $\coprod_{x \in X} Y$  is simply the binary product of sets  $X$  and  $Y$ .

Let  $A$  and  $B$  be propositions represented by types  $A$  and  $B$ . Then a canonical proof representation of  $A \wedge B$  is a tuple of proofs of  $A$  and  $B$ . This corresponds to the definition of  $\text{And } A B$ , whose canonical proof object is a tuple of proof objects of type  $A$  and type  $B$ . Conversely, if  $\text{ab}$  be a proof object of  $\text{And } A B$ , then  $\text{ab.fst}$  and  $\text{ab.snd}$  are proof objects of type  $A$  and  $B$ , respectively.

### 2.4.3 Logical equivalence.

The statement that propositions  $A$  and  $B$  are logically equivalent, written as  $P \equiv Q$ , is defined as  $(A \supset B) \wedge (B \supset A)$ . The logical constant  $\text{Iff}$  is an `agda`-expression for logical equivalence:

```
Iff (X :: Set) (Y :: Set) :: Set
  = And (Imply X Y) (Imply Y X)
```

## 2.5 Plus type.

The type `Plus` is defined by the data type definition. The plus type represents the binary **disjoint union** in type theory.

```
Plus (X :: Set) (Y :: Set) :: Set
  = data inl (x :: X) | inr (y :: Y)
```

The data type has two unary constructors `inl@_` and `inr@_`, such that `inl@_ x` for  $x :: X$  and `inl@_ y` for  $y :: Y$  are `agda` terms of type `Plus X Y`.

### 2.5.1 Disjunction.

The binary operator `Or` is defined by `Plus`,

```
Or (X :: Set) (Y :: Set) :: Set
  = Plus X Y
```

Let  $A$  and  $B$  be propositions represented by  $A$  and type  $B$ . Clearly any proof of  $A$  or  $B$  is a proof of  $A \vee B$ : Conversely, any canonical proof of  $A \vee B$  is obtained either from a proof of  $A$  or from a proof of  $B$ . This corresponds to the definition of `Or A B`, whose canonical proof object is either `inl@_ a` for  $a :: A$  or `inl@_ b` for  $b$  of type  $B$ . Hence, `Or A B` has a proof object if and only if there is either a proof object  $a$  of type  $A$  or a proof object of type  $B$ . This is called *Disjunction property* in Intuitionistic logic: The Existence property says that if a formula  $A \vee B$  is provable, then either  $A$  or  $B$  is provable.

## 2.6 Equality.

The basic equality relation is the *identity* `Id` in the `SET` library.

```
-----
-- Identity proof sets.
-----
```

```

Id (X::Set) :: Rel X
  = idata ref (x::X) :: _ x x
refId (X::Set) :: Reflexive X (Id X)
  = \ (x::X) -> ref@_ x

```

Id is defined by **idata type construction**. Let  $A :: \text{Set}$ . Then  $\text{Id } A$  has type  $\text{Rel } A$ , which is equal to  $A \rightarrow A \rightarrow \text{Set}$ , as seen in the following definition in the SET library:

```

Rel (X::Set) :: Type
  = X -> X -> Set

```

Here `Type` is one of the type of **sorts** in `agda`. In `agda` type system, any expression is well-typed. The type `Set`, formally `#0`, is the first type of sorts. Since `Set` cannot be typed by `Set` itself, the second type `Type` of sorts, formally `#1`, is required: Thus we have `Set :: Type`. The type `Rel` above is another example of type `Type`.

Let  $a :: A$ . The type  $\text{Id } A \ a \ a$  is minimal in the sense that the only object of type  $\text{Id } A \ a \ a$  is `ref@_ a` obtained by the constructor `ref@_ .`

The laws of equality, namely, the reflexivity, the symmetry and the transitivity are expressed, in the SET library as `Reflexive`, `Symmetrical` and `Transitive`, respectively.

```

Reflexive (X::Set)(R::Rel X) :: Set
  = (x::X) -> R x x
Symmetrical (X::Set)(R::Rel X) :: Set
  = (x1::X) -> (x2::X) -> R x1 x2 -> R x2 x1
Transitive (X::Set)(R::Rel X) :: Set
  = (x1::X) -> (x2::X) -> (x3::X) -> R x1 x2 -> R x2 x3 -> R x1 x3

```

They are not proofs of the laws, but name constants for the statements. In case of the identity `Id`, we can provide direct proofs:

```

--#include "Hedberg/Logic/Set.alfa"

open SET use Reflexive, Symmetrical, Transitive, Id

refId(X::Set)::Reflexive X (Id X) = \ (x::X)-> ref@_ x

symId(X::Set)::Symmetrical X (Id X)
  = \ (x,y::X)-> \ (h::Id X x y)-> case h of (ref z)-> h

trnId(X::Set)::Transitive X (Id X)
  = \ (x,y,z::X)-> \ (xy::Id X x y)-> \ (yz::Id X y z)->
    case xy of (ref x')-> yz

```

One of the useful propositions on `Id` is `mapId` in the SET library.

```

mapId(X, Y :: Set)(f:: X -> Y)
  :: (x1,x2::X) -> Id X x1 x2 -> Id Y (f x1) (f x2)

```

Let  $A$  and  $B$  be sets with the equalities  $=_A$  and  $=_B$ , respectively. Let  $f$  be a function from  $A$  to  $B$ . Then the function  $f$  is called *extensional*, if  $x =_A y$

implies  $f(x) =_B f(y)$ . The `mapId` formulates the extensionality with the identity as the equality. Though `mapId` is shown in the SET library, we show its simplified (and direct) proof here.

```
mapId' (X, Y :: Set) (f :: X -> Y)
  :: (x1, x2 :: X) -> Id X x1 x2 -> Id Y (f x1) (f x2)
  = \ (x1, x2 :: X) -> \ (h :: Id X x1 x2) ->
    case h of (ref x) -> ref@_ (f x)
```

## 2.7 Examples.

We shall explain `agda` proofs of

1. Intuitionistic Propositional logic,
2. Classical Propositional logic,
3. Intuitionistic first order predicate logic,
4. Classical first order predicate logic,
5. First order arithmetic.

### 2.7.1 Intuitionistic Propositional logic.

In propositional logic, we define formulas from propositional variables combined with propositional logical connectives.

We shall demonstrate how to prove the following proposition in `Agda`:

```
Imply (And A B) C 'Iff' Imply A (Imply B C).
```

We note that *back quotation* (`'`) is applied to the binary operator `Iff` to make it an infix one. We often use this convention in order to improve readability.

Let us open a new file and start with opening the SET package.

```
--#include "Hedberg/SET.alfa"
```

```
open SET use Imply, Absurd, Taut, Not, And, Iff, Or
```

To prove the proposition in `Agda` is to construct an object of that proposition seen as a type. We make the goal `0` as follows:

```
prop1 (A, B, C :: Set)
  :: Imply (And A B) C 'Iff' Imply A (Imply B C)
  = {}0
```

The proof now being constructed has its name `prop1` and bound propositional variables `A`, `B`, `C`. According to the discussions in 2.4.2, a proof object of type `And A B` is the pair of proof objects of types `A` and `B`.

As the first step, we recommend the **Intro** command: This command gives us a prototype for the expression of the goal type. Our strategy is based on a search of the normalized proof in natural deduction of the corresponding formula

$$((A \wedge B) \supset C) \equiv (A \supset (B \supset C))$$

in intuitionistic logic.

By invoking **Intro** of the goal command, we obtain this:

```

Prop1 (A, B, C::Set)
  :: Impl (And A B) C 'Iff' Impl A (Impl B C)
  = struct
    fst = {}0
    snd = {}1

```

where the subgoals are of the following types.

```

?1 :: Impl A (Impl B C) -> Impl (And A B) C
?0 :: Impl (And A B) C -> Impl A (Impl B C)

```

We invoke **Intro** and **Solve**, then obtain this:

```

fst = \ (h::Impl (And A B) C) -> {}2

```

Once again, we invoke **Intro** and **Solve**, and obtain this:

```

fst = \ (h::Impl (And A B) C) ->
      \ (h'::A) -> {}3

```

with ?3 :: Impl B C. We can go one step further by **Intro**, but let us examine the definition body obtained so far. We need to prove Impl B C assuming that we have proofs h and h' of Impl (And A B) C and A, respectively. For readability, let us change the automatically chosen name h' to a simply by editing the buffer; however we should let agda know the newly chosen name by invoking **Load**.

The definition of type Impl B C is (non-dependent) function type B -> C. Therefore we need a *function expression*, i.e., the  $\lambda$  term expression here. The **Abstraction** command gives us a prototype for the function expressions of the given type. Type “b” in the goal 3 below<sup>3</sup>.

```

fst = \ (h::Impl (And A B) C) ->
      \ (a::A) -> {}3

```

and invoke **Abstraction**. Then we obtain this:

```

fst = \ (h::Impl (And A B) C) ->
      \ (a::A) -> \ (b::{}4) -> {}5

```

By **Solve** command, we can further fill the goal 4, and we are left with

```

fst = \ (h::Impl (And A B) C) ->
      \ (a::A) -> \ (b::B) -> {}5

```

with ?5 :: C. Let us use h of type Impl (And A B) C, or equivalently And A B -> C. ■

If an expression of type And A B is applied to h, the application expression has type C. Invoke the **Refine** command here to fill the goal 5 with h. The **Refine** checks the types of the goal and the object to fill.

If the two types coincide, the object simply replaces the goal: If they don't, agda provides, if any, an application expression with new subgoals. Since here an application expression with h can have the right type, it is expressed with a newly introduced subgoal 6 of type And A B.

<sup>3</sup>We could type several variables in a goal on the right hand side to obtain a function expression of three these variables.

```
fst = \ (h :: Imply (And A B) C) ->
      \ (a :: A) -> \ (b :: B) -> h {}6
```

To proceed further, invoke **Intro** at goal 6:

```
fst =
  \ (h :: Imply (And A B) C) ->
  \ (a :: A) -> \ (b :: B) ->
    h (struct
      fst = {}7
      snd = {}8
    )
```

The obvious choices of the goals 7 and 8 are **a** and **b**, since a proof object of type **And A B** is a tuple of proofs of type **A** and type **B**. Hence type **a** (and **b**) into the goal 7 (and goal 8, respectively) and invoke **Refine**. It completes the definition of **fst**. Once we complete the definition, i.e., there is no goals left, we can delete unnecessary spaces to make it look better.

```
fst = \ (h :: Imply (And A B) C) ->
      \ (a :: A) -> \ (b :: B) -> h struct{ fst = a; snd = b;}
```

Let us turn to **snd**: We similarly invoke each **Intro** and **Solve** twice, change the default variable name to **ab**: Then we obtain this:

```
snd =
  \ (h :: Imply A (Imply B C)) ->
  \ (ab :: And A B) -> {}9
```

with **?9 :: C**. We use **h** here.

```
snd =
  \ (h :: Imply A (Imply B C)) ->
  \ (ab :: And A B) -> h {}9
                        {}10
```

with **?9 :: A** and **?10 :: B**. Invoke **Infer type (unfolded)** and give **ab** to the prompt, then we can check that the variable **ab** has type **Sum A (\(x :: A) -> B)**. ■

Notice that  $\lambda(x :: A) \rightarrow B$  has type **A -> Set** because **B** has type **Set**. Hence **ab.fst** has type **A** and **ab.snd** has type  $\lambda(x :: A) \rightarrow B$  **fst**, which is equal to **B**. Hence our obvious choices for goals 9 and 10 are **ab.fst** and **ab.snd**.

```
snd =
  \ (h :: Imply A (Imply B C)) ->
  \ (ab :: And A B) -> h ab.fst ab.snd
```

Finally, we show the complete proof.

### 2.7.2 Classical propositional logic.

In this subsection, we shall demonstrate a simple way to prove a tautology in Classical propositional logic using **agda**.

*The law of excluded middle* is a formula  $P \vee \neg P$  with a formula  $P$ . In Classical logic, the law of excluded middle holds for any formula  $P$ . In Intuitionistic logic, the law of excluded middle is provable if the proposition  $P$  is decidable.

```

--#include "Hedberg/SET.alfa"

open SET use Imply, Absurd, Taut, Not, And, Iff, Or

Prop1 (A, B, C::Set)
  :: Imply (And A B) C 'Iff' Imply A (Imply B C)
  = struct
    fst = \(h::Imply (And A B) C)->
          \(a::A)-> \(b::B)-> h struct{ fst = a; snd = b;}
    snd = \(h::Imply A (Imply B C))->
          \(ab::And A B)-> h ab.fst ab.snd

```

Figure 4: Source code 3

As for an example, let “==” be the decidable equality between natural numbers defined in the `OpNat` package. Then  $x == y$  ‘Or’  $\text{Not}(x == y)$  is provable in `agda` without the law of excluded middle.

```

--#include "Hedberg/Op/Nat.alfa"
--#include "Hedberg/Logic/Bool.alfa"

public open SET
  use Unop, Binop, Nat, Bool, True
public open LogicBool
  use true, false, not, or
public open OpNat
  use (==) :: Nat -> Nat -> Bool

lem (x, y::Nat):: True (or (x == y) (not (x == y)))
  = let pem (p::Bool)::True (or p (not p))
      = case p of {(true )-> tt@_ ; (false)-> tt@_ ; }
      in pem (x == y)

```

In general, there are two ways to prove a classical tautology  $A$  in `agda`: One is to prove alternatively  $\neg\neg A$ , known to be logically equivalent to  $A$  itself by Glivenko’s theorem. The other is to define a package, in which desired classical principle, for example the law of excluded middle, is available. In this subsection, we take the latter approach.

We shall demonstrate how to prove the following proposition in `Agda`:

Imply A B ‘Iff’ Or (Not A) B.

Let us open a new file and start with opening the `SET` package. We define package `Classical` starting as follows.

```

--#include "Hedberg/SET.alfa"

open SET use Not, Or
Prop :: Type = Set

```

```
package Classical (em ::(P::Prop) -> Or P (Not P)) where
```

```
  open SET use ImPLY, Absurd, Taut, Not, And, Iff, Or
```

We make the goal 0 as follows:

```
prop2 (A, B :: Prop) :: ImPLY A B 'Iff' Or (Not A) B
= {}0
```

By invoking **Intro** and we obtain the following:

```
prop2 (A, B :: Prop) :: ImPLY A B 'Iff' Or (Not A) B
= struct
  fst = {}0
  snd = {}1
```

Let us work on `fst` at first: By invoking **Intro** and **Solve** we arrive the following:

```
fst = \ (h::ImPLY A B)-> {}2
```

Since we need argue, based on the law of excluded middle, depending on whether `A` has a proof or `Not A` does, we invoke case analysis. Type `em A` and invoke **Case**, and we obtain the following:

```
fst = \ (h::ImPLY A B)->
  case em A of
  (inl x)-> {}3
  (inr y)-> {}4
```

Here `x` and `y` represent proofs of `A` and `Not A`, respectively.

Thus we replace them by new names `a` and `na`, and `a` and `na` have types `A` and `Not A`, respectively.

```
fst = \ (h::ImPLY A B)->
  case em A of
  (inl a)-> {}3
  (inr na)-> {}4
```

Let us work on the goal 3: Notice that it has type `Or (Not A) B`. Since `h` is a function of type `A -> B`, the application `h a` has type `B`. Then `inr@_ (h a)` has type `Or (Not A) B`, which we were looking for.

```
fst = \ (h::ImPLY A B)->
  case em A of
  (inl a)-> inr@_ (h a)
  (inr na)-> {}4
```

Similarly the goal 4 is filled with `inl@_ na`:

```
fst = \ (h::ImPLY A B)->
  case em A of
  (inl a)-> inr@_ (h a)
  (inr na)-> inl@_ na
```

Let us now turn to `snd = {}1` in the initial stage. By invoking **Intro** and **Solve** a few times, we obtain the following, where bound variable `a` of type `A` is preferably introduced:

```
snd = \ (h :: Or (Not A) B) -> \ (a :: A) -> {}5
```

Since the proof  $h$  of  $\text{Or (Not A) B}$  has **Plus** type, we are to argue by cases whether the  $h$  comes from the proof of  $\text{Not A}$  or  $B$ . So, type “ $h$ ” in the goal 5 and invoke **Case**:

```
snd = \ (h :: Or (Not A) B) ->
      \ (a :: A) -> case h of (inl x) -> {}6
                          (inr y) -> {}7
```

In the case  $\text{inl } x$  ( $\text{inr } y$ ), the  $h$  comes from the proof  $x$  ( $y$ ) of  $\text{Not A}$  ( $B$ , respectively). The goal 7 of type  $B$  is easily filled with  $y$  itself.

```
snd = \ (h :: Or (Not A) B) ->
      \ (a :: A) -> case h of (inl x) -> {}6
                          (inr y) -> y
```

As for the goal 6, notice that  $x$  has type  $\text{Not A}$ : Since  $a$  has type  $A$ , the expression  $x \ a$  has type **Zero**, the empty set, which can be easily checked by invoking **Infer type**. Once an expression of type **Zero** is obtained, we invoke **Case** to successfully fill the goal as follows:

```
snd = \ (h :: Or (Not A) B) ->
      \ (a :: A) -> case h of (inl x) -> case x a of { }
                          (inr y) -> y
```

The last step corresponds to the *Absurdity elimination*: Any proposition  $A$  follows from the Absurdity  $\perp$ . This means that a proof object  $x \ a$  of the Absurdity type is also a proof object of arbitrary type.

Finally, we show the complete proof:

```
--#include "Hedberg/SET.alfa"

open SET use Not, Or
Prop :: Type = Set

package Classical (em :: (P :: Prop) -> Or P (Not P)) where

  open SET use Imply, Absurd, Taut, Not, And, Iff, Or

  prop2 (A, B :: Prop) :: Imply A B 'Iff' Or (Not A) B
  = struct
    fst = \ (h :: Imply A B) ->
          case em A of {(inl a) -> inr@_ (h a); (inr na) -> inl@_ na;}
    snd = \ (h :: Or (Not A) B) -> \ (a :: A) ->
          case h of {(inl x) -> case x a of { }; (inr y) -> y;}
```

Figure 5: Source code 4

### 2.7.3 Intuitionistic first-order predicate logic.

In this subsection, we shall prove a proposition in Intuitionistic first-order predicate logic.

Let  $P$  be a formula without free variable  $x$ . We shall demonstrate how to prove the logical equivalence between  $\exists xP$  and  $P$ . Let  $D :: \text{Set}$  and  $P :: \text{Pred } D$ . The expression `Exist D P` corresponds to  $\exists xP$  in `agda`, where the set  $D$  is used as the domain of individuals for the first-order predicate logic.

Although we can use the `SET` package as before, we use `LogicSet` package, where `Prop` is defined as `Set`.

```
--#include "Hedberg/Logic/Set.alfa"
```

```
Prop :: Type
      = Set
```

The above proposition is written in Agda as follows:

```
pred3 (P::Prop) :: Exist D (\(x::D)-> P) 'Iff' P
      = ?
```

A formula  $P$  not containing free variable  $x$  is expressed in Agda as the function expression  $\lambda(x::D). P$  of type  $D \rightarrow \text{Set}$ , which stands for a constant function.

Now here we need to be careful: This proposition is shown true only when the domain is non-empty. The non-emptiness assumption of the domain is attained by introducing an additional parameter to the package. So let us begin with a new package called `predicate`:

```
package predicate(D::Set)(d0::D) where
  open LogicSet use Prop, Pred, Or, And, Forall, Not, Exist, Iff
  Prop :: Type
        = Set
```

```
pred3 (P::Prop) :: Exist D (\(x::D)-> P) 'Iff' P
      = ?
```

Invoking `Intro` and `Solve` a few times, we obtain the following:

```
pred3 (P::Prop) :: Exist D (\(x::D)-> P) 'Iff' P
      = struct
        fst = \(h::Exist D (\(x::D)-> P))-> {}0
        snd = {}1
```

The definition of `fst` corresponds to a proof of `Exist D (\(x::D)-> P) 'Imply' P`.

The variable `h` has type `Exist D (\(x::D)-> P)`, which is unfolded to `Sum D (\(x::D) -> P)`.

Due to the definition of `Sum` type, `h.fst` has type  $D$  and `h.snd` has type  $(\lambda(x::D). P)$  `h.fst`, which is  $\beta$ -reduced to  $P$ .

Since the goal `0` has type  $P$ , type “`h.snd`” in the goal `0` and Invoke `Refine`: This gives us the following:

```
fst = \(h::Exist D (\(x::D)-> P))-> h.snd
```

Now let us work on `snd`: The definition body of `snd` corresponds to a proof of `P 'Imply' Exist D (\(x::D)-> P)`. Again we can obtain the following without any difficulty:

```
snd = \ (h::P)-> struct
      fst = {}1
      snd = {}2
```

Here the new goal 1 has type D. Only expression of type D available to us at the moment is `d0` we assumed in the package. Hence we completed the definition as follows:

```
fst = d0
```

Since the goal 2 has type P, it is filled with `h` itself:

```
snd = h
```

Finally, we show the complete proof:

```
--#include "Hedberg/Logic/Set.alfa"
package predicate(D::Set)(d0::D) where
  open LogicSet use Prop, Pred, Or, And, Forall, Not, Exist, Iff

  Prop :: Type = Set

  pred3 (P::Prop) :: Exist D (\(x::D)-> P) 'Iff' P
  = struct
    fst = \ (h::Exist D (\(x::D)-> P))-> h.snd
    snd = \ (h::P)-> struct {fst = d0; snd = h;}
```

Figure 6: Source code 5

#### 2.7.4 Classical first-order predicate logic.

We shall prove classically a proposition

$$\forall x(P \vee Q(x)) \implies (P \vee \forall Q(x)),$$

where  $P$  does not contain free variable  $x$ . Since we need law of exclusive middle, we shall begin with a new package `ClassicalPred`.

```
--#include "Hedberg/Logic/Set.alfa"

package ClassicalPred(D::Set)
  (d::D)
  (em :: (P::Prop)-> P 'Or' Not P) where

  open LogicSet use Prop, Pred, Or, And, Forall, Not, Exist, Iff

  pred4 (P::Prop)(Q::Pred D)
  :: Forall D (\(x::D)-> P 'Or' Q x) 'ImPLY' Or P (Forall D Q)
  = {}0
```

By invoking **Intro** and **Solve** a few times, we obtain the following:

```

pred4 (P::Prop)(Q::Pred D)
  :: Forall D (\(x::D)-> P 'Or' Q x) 'ImPLY' Or P (Forall D Q)
  = \(h::Forall D (\(x::D)-> Or P (Q x)))-> {}1

```

Let us give an informal proof of the proposition at first. We need to prove  $P \vee \forall Q(x)$  under the assumption of  $\forall x(P \vee Q(x))$ . We argue by cases whether proposition  $P$  holds or not in the model with the universe  $D$ . If  $P$  holds in the model, then so does  $P \vee \forall Q(x)$ . If  $P$  does not hold in the model, then  $\forall xQ(x)$  must hold since  $\forall x(P \vee Q(x))$  holds. Thus we again obtain  $P \vee \forall Q(x)$ .

Now we encode this proof in `agda`: Type “em P” and invoke **Case**. This provides us the following:

```

pred4 (P::Prop)(Q::Pred D)
  :: Forall D (\(x::D)-> P 'Or' Q x) 'ImPLY' Or P (Forall D Q)
  = \(h::Forall D (\(x::D)-> Or P (Q x)))->
    case em P of
      (inl x)->{}2
      (inr y)->{}3

```

Now `inl x` (`inr y`) represents the proof of `Or P (Not P)` obtained from the proof `x` (`y`) of `P` (`Not P`, respectively). Thus we replace them by new constant names `p` and `np`, of types `P` and `Not P`, respectively.

```

pred4 (P::Prop)(Q::Pred D)
  :: Forall D (\(x::D)-> P 'Or' Q x) 'ImPLY' Or P (Forall D Q)
  = \(h::Forall D (\(x::D)-> Or P (Q x)))->
    case em P of
      (inl p)->{}2
      (inr np)->{}3

```

The goal 2 can be easily filled with `inl@_ p`, since `p` and `inl@_ p` have types `P` and `P 'Or' (Forall D Q)`, respectively.

```

pred4 (P::Prop)(Q::Pred D)
  :: Forall D (\(x::D)-> P 'Or' Q x) 'ImPLY' Or P (Forall D Q)
  = \(h::Forall D (\(x::D)-> Or P (Q x)))->
    case em P of
      (inl p)-> inl@_ p
      (inr np)->{}3

```

As for the goal 3 of type, we need to define an object, say `lem` here, of type `Forall D Q`, which is unfolded to the function type `(x::D) -> Q x`. Type `lem` in the goal 3 and invoke **Let**.

```

pred4 (P::Prop)(Q::Pred D)
  :: Forall D (\(x::D)-> P 'Or' Q x) 'ImPLY' Or P (Forall D Q)
  = \(h::Forall D (\(x::D)-> Or P (Q x)))->
    case em P of
      (inl p)-> inl@_ p
      (inr np)->
        let lem::{}1
          = {}2
        in {}3

```

We need to a little trick here: In stead of giving the type  $(x::D) \rightarrow Q\ x$  of the variable `lem`, we should edit the let expression and get the following one:

```
let lem (x::D)::{}1
    = {}2
in {}3
```

Invoke **Load**, and the variable `x` is added to the context list. Now type “`Q x`” to the goal 1 and invoke **Refine**.

```
let lem (x::D):: Q x
    = {}2
in {}3
```

Since `h` has type `forall D (\(x::D)-> Or P (Q x))`, unfolded to  $(x::D) \rightarrow Or\ P\ (Q\ x)$ , `h x` has type `Or P (Q x)`. Now we argue by cases of `h x`, namely either it comes from `P` or `Q x`. Type “`h x`” and invoke **Case**:

```
let lem (x::D):: Q x
    = case h x of
      (inl x')->{}4
      (inr y )->{}5
in {}3
```

`x'` has type `P`, and so `np x'` has type `Zero`, or the empty set. Thus type “`np x'`” in the goal 4 and invoke **Case**, which completes the object:

```
(inl x')-> case np x' of { }
(inr y )->{}5
```

The variable `y` has type `Q x`, and it fits to the goal 5. So type “`y`” in the goal 5, and invoke **Refine**: This completes the object.

```
(inl x')-> case np x' of { }
(inr y )-> y
```

Since `lem` has type `Q x`, `inr@_ lem` does `Or P (Q x)`. Finally type “`inr@_ lem`” in the goal 3 and invoke **Refine**.

```
let lem (x::D):: Q x
    = case h x of
      (inl x')-> case np x' of { }
      (inr y )-> y
in inr@_ lem
```

This completes the proof.

Finally, we show the complete proof:

### 2.7.5 First-order Arithmetic.

In this subsection, we shall prove the associativity of addition in the first-order arithmetic. In the SET library, the set `Nat` of natural numbers, the zero `zero`, the successor function `succ` are defined:

```

--#include "Hedberg/Logic/Set.alfa"

open LogicSet use Prop, Or, Not

package ClassicalPred(D::Set)
    (d::D)
    (em ::(P::Prop)-> P 'Or' Not P) where

open LogicSet use Prop, Pred, Or, And, Forall, Not, Exist, Iff

pred4 (P::Prop)(Q::Pred D)
  :: Forall D (\(x::D)-> P 'Or' Q x) 'Imply' Or P (Forall D Q)
  = \(h::Forall D (\(x::D)-> Or P (Q x)))->
    case em P of
    (inl p)-> inl@_ p
    (inr np)->
      let lem (x::D):: Q x
          = case h x of {(inl x')-> case np x' of { }; (inr y )-> y;}
      in inr@_ lem

```

Figure 7: Source code 6

```

-----
-- Natural numbers.
-----

Nat :: Set
  = data zer | suc (m::Nat)
zero :: Nat
  = zer@_
succ (x::Nat) :: Nat
  = suc@_ x

```

Now let us examine the addition on the natural numbers defined in a library Hedberg/Op/Nat.alfa.

```

(+) (m::Nat)(n::Nat) :: Nat
  = case m of {
    (zer) -> n;
    (suc m') -> suc@_ ((+) m' n);}

```

The addition + is an infix operator, due to the parenthesis, and is defined recursively on the first parameter. We shall prove the associativity of the addition. In order to express an equation between two expressions of type Nat, we shall use the identity in Section 2.6. Note that the expression `Id Nat a a` corresponds to the proposition that `a` equals to `a` for any object `a` of type `Nat`, and the canonical object `ref@_ a` of type `Id Nat a a` represents the proof object of this type.

Open a new file and begin with a new package `NatAssoc`.

```

--#include "Hedberg/Op/Nat.alfa"

```

```

package NatAssoc where
  public open SET
    use Nat, zero, Id, mapId, succ

  public open OpNat
    use (+) :: Nat -> Nat -> Nat

  ass_add' (x, y, z::Nat) :: Id Nat ((x + y) + z) (x + (y + z))
    = {}0

```

It is hard to predict which functions will be used to write the definition body of `ass_add'`. We generally discover them in due course. Here we have imported them in advance to make our explanation simpler.

Remember that the addition `+` is defined on the first parameter. We shall construct the definition body of `ass_add'` inductively on `x`. In `agda`, such an induction principle is achieved by typing “`x`” in the goal 0 and invoking **Case**:

```

ass_add' (x, y, z::Nat) :: Id Nat ((x + y) + z) (x + (y + z))
  = case x of
    (zer )-> {}1
    (suc m)-> {}2

```

In the case of `x` has value `zer`, the expressions `(x + y) + z` and `x + (y + z)` have the same value `y + z`. So type “`ref@_ (y + z)`” to the goal 1, and invoke **Refine**:

```

(zer )-> ref@_ (y + z)

```

Now let us turn to the goal 2. The expressions `((suc m) + y) + z` and `(suc m) + (y + z)` are evaluated to `suc@_ (m + y + z)`<sup>4</sup> and `suc@_ (m + (y + z))`, respectively.

To prove that these expressions have the same value, we shall use the induction hypothesis (`ass_add' m y z`); and the proposition `mapId` (See, Section 2.6 for the details) applied to the successor function `succ :: Nat -> Nat`, that if two expressions of type `Nat` have the same value, then so do their successors.

Finally, we show the complete proof:

## 2.8 Exercises.

Prove the following using in *agda*.

1. Intuitionistic Propositional logic.

(a) `Exer1 (A::Set) :: A 'ImPLY' (Not (Not A))`

(b) `Exer2 (A,B::Set) :: Or (Not A) B 'ImPLY' ImPLY A B`

(c) `Exer3 (A,B::Set) :: And (Not A) (Not B) 'ImPLY' Not (Or A B)` ■

---

<sup>4</sup>There is a fixed set of operator precedences in `agda`, (refer to Coquand [2]): The operator `+` is left-associative of precedence 6.

```

--#include "Hedberg/Op/Nat.alfa"

package NatAssoc where
  public open SET
    use Nat, zero, Id, mapId, succ

  public open OpNat
    use (+) :: Nat -> Nat -> Nat

  ass_add' (x, y, z::Nat) :: Id Nat ((x + y) + z) (x + (y + z))
  = case x of
    (zer _) -> ref@_ (y + z)
    (suc m) -> mapId Nat Nat succ (m + y + z)
              (m + (y + z))
              (ass_add' m y z)

```

Figure 8: Source code 7

(d) (Hint: Use **Abstraction**.)

Exer4 (A,B::Set) :: And (Not A) (Not B) 'ImPLY' Not (Or A B) ■

(e) Exer5 (A,B::Set) :: ImPLY A B 'ImPLY' ImPLY (Not B) (Not A) ■

2. Classical Propositional logic.

(a) Exer6 (A::Set) :: Not (Not A) 'ImPLY' A

(b) Exer7 (A::Set) :: ImPLY A B 'ImPLY' Or (Not A) B

3. Intuitionistic Predicate logic.

(a)  $\exists x(P \wedge Qx) \equiv P \wedge \exists xQx$ , if variable  $x$  is not free in  $P$ .

Exer8 (P::Prop)(Q::Pred D) ::  
Exist D (\(x::D)-> P 'And' Q x) 'Iff' And P (Exist D Q) ■

(b)  $\exists xPx \equiv \exists yPy$ . (Change of bound variables).

Exer9 (P::Prop) :: Exist D (\(x::D)-> P) 'Iff' Exist D (\(y::D)-> P) ■

(c)  $P \vee \exists xQx \equiv \exists x(P \vee Qx)$ , if variable  $x$  is not free in  $P$ .

Exer10 (Q::Pred D) (P::Prop) ::  
Or P (Exist D Q) 'Iff' Exist D (\(y::D)-> (Or P (Q y))) ■

(d)  $(P \vee \forall xQx) \supset \forall x(P \vee Qx)$ , if variable  $x$  is not free in  $P$ .

Exer11 (Q::Pred D)(P::Prop) ::  
 Or P (Forall D Q) 'Imply' Forall D (\(y::D)-> (Or ((\ (x::D)-> P) y) (Q y)))■

(e)  $\forall x Px \wedge \forall x Qx \equiv \forall x (Px \wedge Qx)$ .

Exer12 (P::Pred D)(Q::Pred D) ::  
 And (Forall D P) (Forall D Q) 'Iff' Forall D (\(y::D) -> (And (P y) (Q y)))■

(f)  $\forall x Px \vee \forall x Qx \supset \forall x (Px \vee Qx)$ .

Exer13 (P::Pred D)(Q::Pred D) ::  
 Or (Forall D P) (Forall D Q) 'Imply' Forall D (\(y::D) -> (Or (P y) (Q y)))■

#### 4. Classical Predicate logic.

(a)  $\forall (P \vee Qx) \supset P \vee \forall x Qx$ , if free variable  $x$  does not occur in  $P$ .

Exer14 (P::Prop)(Q::Pred D) ::  
 Forall D (\(x::D)-> P 'Or' Q x) 'Imply' Or P (Forall D Q)■

## References

- [1] Nordström, B., Petersson, K. and Smith, J.M., *Programming in Martin-Löf's Type Theory*, <http://www.cs.chalmers.se/Cs/Research/Logic>.
- [2] Coquand, C., *Syntax in Agda documentation*, <http://www.cs.chalmers.se/catarina/agda/syntax.html>.