An Agda Tutorial

Misao Nagayama
`m-nagayama@aist.go.jp`

Hideaki Nishihara
`hide.a.kit@ni.aist.go.jp`

Makoto Takeyama
`makoto.takeyama@aist.go.jp`

Research Center for Verification and Semantics (CVS)
National Institute of Advanced Industrial Science and Technology (AIST)
Japan

May 12, 2006

Contents

1 Introduction.

1.1 What is agda?

[To be written.]

1.2 A brief history of agda

[To be written.]

1.3 Who reads this document?

With this document we will know how to use `Agda`: operations, its language, and typical ways to define functions. One of important goals of this document is to write proofs. Features and functions listed above and many examples will help readers to implement a formal system and to prove theorems on the formal system.

This tutorial is for the people who have some interests in `Agda` but who have not used `Agda`. They may be interested in pure/applied logics, programming with types, differences with other theorem-provers/proof-assistants, `Agda` itself, and so on.

Readers are supposed to have some knowledge and experiences on:

- Classical logic and intuitionist logic, and natural deduction;
- Simple Type Theory;
- Typed lambda calculi;
- The Emacs editor.

Moreover they will progress more efficiently with the following knowledge and experiences:

- Functional programming languages, especially Haskell;
- Dependent type theory, especially Martin Löf Type theory;
- Curry-Haward correspondence or "Propositions as sets" paradigm.

There are still some features of `Agda` we do not deal with in this document, and detailed explanations are often omitted in this document. When readers want advanced informations, reference manual will help them. (But in preparation.)

1.4 Notations

In this tutorial, `Agda` denotes the system consisting of a typechecker, standard libraries, and interfaces on Emacs editor.

Descriptions in this tutorial are based on the version released on 2005 April[1].

It is remarked that `Agda` is now a developing system. Different behaviors, and different message outputs may be seen in case different versions of `Agda` are used. Especially specifications on hidden arguments (Section 3.13) may change.

[1] `http://coverproject.org/Agda/Agda-1.1-cvs20050411.tar.gz`

Typefaces. Strings in special typefaces have special meanings. Words and lines in the `teletype` typeface are filenames, parts of codes, strings to input, or ones about the computer environment[2]. A string in angle brackets like as "$\langle$*foo*$\rangle$" is to be replaced with appropriate expressions in a code. What expression is fit for a $\langle$*foo*$\rangle$ is shown each time. Keywords and `Agda` commands are printed in the **bold** face.

An abstract expression and its implementation are distinguished by their typefaces in this document. For example, consider the following identifiers in the mathematical world:

> a set A, a function $f : A \to A$, and an element a of A.

When discussing about their implementations on `Agda`, they are referred to:

> a set(type) `A`, a function `f` of type `A -> A`, and an element(object) `a` of `A`.

Key inputs. As in many documents on Emacs, `C-`, `M-`, and `RET` mean "Control-", "Meta-", and "Enter" respectively. For example, `C-c` means to input '`c`' with Control key.

Key binding for `Agda` commands are printed in a box like as $\boxed{\texttt{C-c C-q}}$. It is a sequence of inputs: `C-q` following `C-c`.

1.5 Acknowledgement

The authors are grateful to the developers of `Agda`, in particular the people who joined the 2nd `Agda` Meeting: Andreas Abel, Marcin Benke, Ana Bove, Catarina Coquand, Thierry Coquand, Peter Dybjer, Bengt Nordström, and Ulf Norell. Discussions with them helped us to learn the theoretical background, implementations and applications of `Agda`. Also with their comments, the authors could improve this tutorial on exactness and on readability.

The authors are grateful to members in CVS/AIST, in particular who participated in the 2nd `Agda` Meeting and who joined to an internal seminar of `Agda`. They gave us valuable comments for this tutorial.

This tutorial is supported in part by Core Research for Evolutional Science and Technology (CREST) Program "New High-performance Information Processing Technology Supporting Information-oriented Society" of Japan Science and Technology Agency(JST).

[2] There is an exception. We use the teletype typeface for referring the system `Agda`.

1.6 New Features

In the summer of 2005, `Agda` is modified in several points. Here we introduce differeces between descriptions in this tutorial and the new version of `Agda`. If the reader uses the latest versions of `Agda` (including 'release versions'), (s)he should visit this section frequently and modify example codes in the later sections. Translations of programs appearing in this tutorial into the new syntax are placed Appendix B.

1.6.1 Syntax

New data type definition. [**C.f. Sect. 3.1, Sect. 3.11**] A data type definition ONLY allows `data` at the beginning of the definition.

```
data Bool'::Set = true | false
data List' (X::Set) = nil | con (x::X) (xs::List' X)
```

No '`@_`' symbols for the constructors. [**C.f. Sect. 3.11, Sect. 4**] Constructor expressions (⟨*Constructor_name*⟩`@_` ⟨*e 1*⟩ ⟨*e 2*⟩ ...) are now prohibited. Corresponding functions (automatically defined by data declarations) must be used. For example,

```
con@_ (con@_ x nil@_) nil@_
```

in the OLD syntax should be written as

```
con (con x nil) nil
```

in the new syntax.
Remarks.

- As a consequence, distinct data types in a same scope must have distinct constructor names. Moreover, we must open EXPLICITLY functions corresponding constructors in a package.
- `Agda`'s outputs may be old expression. That is, the symbol '`@`' appears in computation results.

Left parameters are hidden. [**C.f. Sect. 3.2, Sect. 3.13**] In a definition, the parameters that are declared between the identifier and the double colon are treated as hidden arguments.

The following definition (under the assumption the type `Nat` is defined)

```
succ (n::Nat)::Nat = suc n
```

is equivalent to the following definition in the OLD syntax.

```
succ (|n::Nat)::Nat = suc n
```

Hence when the function `succ` is applied to an expression, the vertical bar must be placed:

```
succ one
```

is to be modified to

```
succ |one
```

In order to switch a hidden parameter explicit, we must place the symbol '`!`' on the left of the parameter. The following definition

```
succ (!n::Nat)::Nat = suc n
```

is equivalent to the following definition in the OLD syntax.

```
succ (n::Nat)::Nat = suc n
```

Explicit declaration for an equational definition. [**C.f. Sect. 3.2.5**] An equational definition should be accompanied with explcit type declaration. For example (the types `List'` and `Bool'` are defined above)

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

New `idata` definition. [**C.f. Sect. 3.12**] In the new syntax, `ListN`, the type of lists of fixed length, is defined as follows:

```
idata ListN (A::Set) :: Nat -> Set where{
    nil                             :: ListN A zero;
    con(n::Nat)(a::A)(as::ListN A n):: ListN A (succ n);
}
```

The differences from the old syntax are as follows:

- `idata` definitions are already not right hand side values.
- Definitions including the string '`::_`' are not allowed. The type identifier following a constructor and a double colon ('`ListN`' in the 2nd/3rd line) must be written explicitly.
- The type of inductive family of types being defined must be annoted as a function type explicitly. The following code

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

  idata Id (A::Set) :: Rel A where{
      ref (x::A) :: Id A x x

  }
  ```

  occurs an error. We must modify like as

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

  idata Id (A::Set) :: A -> A -> Set where{
      ref (x::A) :: Id A x x

  }
  ```

1.6.2 Commands

Compute Command. [**C.f. Sect. 3.2.4, Appendix A**] The command **Compute** ($\boxed{\texttt{C-c C-x >}}$) is added to Agda menu. Different from old commands like **Compute to depth**, it doesn't need any goal. Just it prompts an expression to compute, and prints the result. However this command works only in top-level. Constants defined locally and variables declared inside a definitions are not recognized.

Disabled Commands. [**C.f. Sect. Appendix A**] The following commands are disabled.

- **Chase-import**
- **Solve**, **Solve Constraint**
- **Unfold constraint**
- **Auto**
- **Compute to depth**, **Compute to depth 100**,
- **Compute WHNF**, **Compute WHNF strict**
- **Continue one step**, **Continue several steps**
- **Unfold one**

1.6.3 Libraries

[**C.f. Section 4**]

- Standard libraries for the new syntax have NOT been released yet.
- Standard libraries are not bundled with `Agda` by default. We can obtain them with CVS (Concurrent Versions System). In detail, check Cover Project site:

  ```
  http://coverproject.org/AgdaPage/download.html
  http://coverproject.org/AgdaPage/
  ```

  Note that the project name is "`Agdalib`", and thus we should use the command '`cvs get Agdalib`' instead of '`cvs get Agda`'.

2 A minimal set of commands – Start, load, and quit

First we learn several basic commands. They let us communicate with `Agda` in babble, that is, we can get `Agda` to typecheck a program, and to memorize it. Although a few `Agda` codes are shown here, their meanings are not explained in this section. We should concentrate on operations and commands here.

Start Start up Emacs editor. We can start `Agda` by opening[3] a file with suffix "`.agda`" or "`.alfa`" in an Emacs buffer. Here we open a new buffer named "`first.agda`."

When `Agda` starts, we can see two windows in a Emacs frame and the "Agda" menu in the Emacs menu bar (Figure 1). The larger window is called **main window**, in which we edit `Agda` codes. The smaller window is called **sub window**, in which informations on `Agda` programs are displayed.

```
| File  Edit  Option  Buffers  Tools  Agda  Help
|
|
|
|-EEE:---F1  Emacs: first.agda       (Agda:run Ind)--L1--All-------
|Agda idata  --- the version info brought to you by Ilya.
|
|[ghc602; built Apr 26 2004 10:43:28]
|-EEE:**-F1  Emacs: * Agda version *    (Agda:run Ind)--L1--All----
|
```

Figure 1: Starting `Agda`

Quit and restart To quit `Agda`, we should use the **Quit** command `C-c C-q`[4]. It quits the proof engine, and tries closing all buffers `Agda` can control, including

- a buffer to display outputs from `Agda`, like as contents in the sub window
- a buffer consisting of `Agda` codes[5].

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`.

Load Start `Agda` again (if needed), and try writing the following line in the main window:

```
data Bool' = true | false
```

[3] There is another way: changing Emacs major mode to **agda-mode** by the command `M-x` followed by '`agda-mode`'.

[4] We can invoke **Quit** command in `Agda` menu. Each command in this tutorial can be invoked by the same manner, but we do not mention it at each time.

[5] More precisely, every buffer whose major mode is "`Agda`".

Now we must save the code as a file, as `Agda` checks the existence of the file. We name it "`first.agda`"

To make `Agda` load this program, invoke **Chase-Load** command[6] by `C-c C-x RET`. By invoking **Chase-Load** command, `Agda` loads and typechecks a program, memorizes definitions in it, and shows some informations about it. But the program we have input is so simple that few changes occur in Emacs. Indeed, all changes we can see are in the status bar under the main window:

```
| File  Edit  Option  Buffers  Tools  Agda  Help
|
|Bool'::Set = data true | false
|
|-EEE:---F1  Proof: first.agda       (Agda:run Ind)--L1--All-------
|Agda with idata and implicit arguments
|
|[ghc602; built Apr 23 2005 10:43:28]
|-EEE:**-F1  Emacs: * Agda version *    (Agda:run Ind)--L1--All----
|marking goals: first.agda... done
```

The string '`Proof`' in the bar indicates that the status of the main window is "**Proof state**". In Proof state, we can build a code interactively, as will be explained in Section 5. On the other hand, the state we were in before loading the code is called "**Text state**." To go back to **Text state**, we should use the command `C-c '`.

Comments Here we introduce comments in `Agda` programs. 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 "`-}`".

Example program We will end this section by showing an `Agda` program as an example.

```
data List' (X::Set) = nil | con (x::X) (xs::List' X)

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)
```

[6] There are two commands to load a code: **Load** and **Chase-Load**. In this tutorial only **Chase-Load** is used, for its function includes that of **Load**. (See Commands List in Appendix A)

Exercises.

1. Try operations introduced in this section several times.
   (a) Open "`first.agda`" to a Emacs buffer.
   (b) Typecheck "`first.agda`" with **Chase-Load** command.
   (c) Restart `Agda`.
   (d) Quit `Agda`.
2. Typecheck the program shown at the last of this section. (Hint:Indentation is a part of the syntax in `Agda`.(See Section 3.3.2.))

3 Introduction to the Agda language

Here we learn how to build an Agda program, in particular how to build a function. First, we define data types. Data types are not necessary to define functions for example to define an identity function

```
identity (A::Set)::A->A = \(x::A)->x
```

does not need any data types. However data types and their objects make examples concrete.

In this section we deal with elementary data types: Bool' consisting of boolean values and Nat consisting of natural numbers. Next we deal with functions on those types. We introduce how to define functions and how to get a function value. Case expressions are powerful tools to define functions in Agda. We study about case expressions, including recursive functions.

After the above basic expressions, we deal with expressions related to Dependent type theory. Typing rules, families of types, and how to hide arguments trivially inferred are explained.

3.1 Data types

In Agda, not so many data types are prepared as built-in types. Hence we need to define data types on scratch. They will be used in the whole of this section.

Summary

- Examples of data type definitions.
- Typechecking examples.
- Explanation.
- Syntax of data type definitions.

We start with examples. We review that examples and check they are type-checked by Agda. Do not stop if we have some questions in examples, and answers may be given in Explanation parts.

3.1.1 Examples

To define a data type, it is enough to show all objects that belong to that type. Here is the first example.

```
data Bool' = true | false
```

This line is interpreted as "the constant named Bool' is a data type consisting of two objects named true and false respectively." The identifiers true and false are called **constructors** of Bool' type, since they form objects of type Bool'.

In the next example, the type Nat is defined. It has objects with an argument:

```
data Nat = zer | suc (m::Nat)
```

```
| File  Edit  Option  Buffers  Tools  Agda  Help
|
|
|data Bool' = true | false
|
|data Nat =  zer | suc (m::Nat)
|
|
|
|-EEE:---F1  Proof: Intro.agda       (Agda:run Ind)--L1--All-------
|Agda idata  --- the version info brought to you by Ilya.
|
|[ghc602; built Apr 26 2004 10:43:28]
|-EEE:**-F1  Emacs: * Agda version *    (Agda:run Ind)--L1--All----
| marking goals: Intro.agda... done
```

Figure 2: Typecheking succeeds

This line means that "the constant named `Nat` is a data type with exactly two ways to construct its objects: `zer` is an object of `Nat`, and, for any object `m` of `Nat`, `suc m` is an object of `Nat`. Thus objects of `Nat` are

`zer`, `suc zer`, `suc (suc zer)`, `suc (suc (suc zer))`, ...

Here the constructors of the type `Nat` are `zer` and `suc`.

3.1.2 Typecheck

Let us check that the examples above are accepted by `Agda`. Start Emacs and start `Agda` (See the previous section).

We will write all example codes in this section in the same file Its file name is "`Intro.agda`."

Open a new file, input and save the following code, and at last invoke **Chase-Load** command by `C-c C-x RET`.

```
data Bool' = true | false

data Nat =  zer | suc (m::Nat)
```

We can see that typechecking succeeds by the message in the minibuffer. (Figure 2)

3.1.3 Syntax

The syntactical structure for a data type definition is as follows:

`data` ⟨*Tid*⟩ `=` ⟨*Cons 1*⟩ `|` ⟨*Cons 2*⟩ `|` ... `|` ⟨*Cons k*⟩

⟨*Tid*⟩ is the identifier of the type and each ⟨*Cons i*⟩ is a constructor (with arguments):

$\langle Cons\, i\rangle$ is $\langle Id\, i\rangle$, or
$\langle Id\, i\rangle$ $\langle Args\, i\rangle$ and
$\langle Args\, i\rangle$ is $(\langle Vi1\rangle :: \langle Ti1\rangle)$ $(\langle Vi2\rangle :: \langle Ti2\rangle)$ ... $(\langle Vin'\rangle :: \langle Tin'\rangle)$

Here each $\langle Id\, i'\rangle$ [resp. $\langle V i'j'\rangle$ and $\langle T\ i'j'\rangle$] is the identifier of the corresponding constructor [resp. argument and its type].

Remark. When adjacent arguments are of the same type, the notations like as `(x,y::Nat)` are allowed.

3.1.4 Type notations

Here we leave the main topic in this subsection and introduce general syntactical notations in `Agda`.

A declaration that an expression (an identifier, a variable, etc.) $\langle Expr\rangle$ is of type $\langle Type\rangle$ is written in `Agda` as follows:

$\langle Expr\rangle :: \langle Type\rangle$

3.1.5 Identifiers

Identifiers are lexically sequences of letters, digits, and single quote (`'`), where the first characters of them must be letters. There are remarks:

- Single quote has no special role. The identifiers "`m`" and "`m'`" are independent of each other (unless we bind `m'` to an expression depending on `m`).
- `Agda` distinguishes between upper/lower cases, and letters in each case have no special role[7].

3.1.6 Exercises

1. Define a data type '`Int`' consisting of pairs of two natural number, with the constructor '`I`'. (The type `Int` represents a type of integers. We will define operations and an equality on the type, and it will be natural to regard `Int` as a type of integers. This exercise is the first step.)
2. A list of `Nat` is either
   - the empty list, denoted by `nilN`, or
   - a pair consisting of an object of `Nat` and a list of `Nat` with the constructor `conN`.

   Define the type `ListNat` consisting of lists of `Nat`.

[7] C.f. In Haskell, `foo` must be an identifier of a function and `Foo` must be an identifier of a type or a module.

3.2 Functions

There are two ways to define a function (but they are equivalent) in `Agda`. The first way, introduced here, is used in ordinary mathematics. For example:

$f(x) = 3x + 5,$
$\mathrm{sgn}(x) = \frac{x}{|x|},$

The arguments (the variable x) appear in the left hand side as well as the right hand side, namely they do not need lambda notations.

The second way to define functions is introduced in Section 3.5. There we will introduce lambda notations.

Summary

- Examples of function definition/function application.
- Typechecking examples.
- Computing `Agda` expressions.
- Type inferring.
- Syntax.

3.2.1 Examples

Let us start with constant functions that have no arguments. In the examples below, the function `zero` and the function `one` always return the constant values respectively.

```
zero::Nat = zer
one::Nat = suc zer
```

The next example is a definition of the function on `Nat`, which increases its argument by one or two:

```
succ (n::Nat)::Nat = suc n
plus2 (n::Nat)::Nat = suc (suc n)
```

The meaning of the line above is clear: it tells "for a given object `n` of type `Nat`, the expression `plus2 n` is an object in `Nat`," and "`plus2 n` is bound to the expression `suc (suc n)` explicitly."

Remark. It may not be clear that `plus2` is of a function type, since in its definition it is typed to `Nat`. However `Nat` is not the type of the constant `plus2` itself, but the type of the expression `plus2 n`. General explanations of function types are shown in Section 3.10, and here we just see the type of `plus2`: it is of type `(n::Nat)->Nat`.

```
|
|data Nat = zer | suc (m::Nat)
|
|
|zero::Nat = zer
|one::Nat = suc zer
|
|
|plus2 (n::Nat)::Nat = suc (suc n)
|three::Nat = plus2 one
|
|
|-EEE:---F1  Proof: Intro.agda       (Agda:run Ind)--L1--All-------
|Agda idata  --- the version info brought to you by Ilya.
|
|[ghc602; built Apr 26 2004 10:43:28]
|-EEE:**-F1  Emacs: * Agda version *    (Agda:run Ind)--L1--All----
| marking goals: Intro.agda... done
```

Figure 3: Typecheking succeeds

3.2.2 Application

A function application is presented by an juxtaposition: for example,

```
plus2 one
plus2 (plus2 one)
```

The expression “`plus2 one`” is an application of the function `plus2` to the expression `one` of type `Nat`. The expression in the second line is an application of the function `plus2` to the expression `plus2 one` of type `Nat`.

3.2.3 Typecheck

Let us check the examples above are accepted by `Agda`. Input the following code at the end of “`Intro.agda`”, and invoke **Chase-Load** command.

```
zero::Nat = zer
one::Nat = suc zer

plus2 (n::Nat)::Nat = suc (suc n)
three::Nat = plus2 one
```

We can see that typechecking succeeds by the message in the minibuffer. (Figure 3)

3.2.4 Computation

We can infer that the expression “`plus2 one`” is reduced to the expression

```
suc (suc (suc zer))
```

Let us check it on `Agda`. But direct interfaces to display reduced forms of expressions are not prepared in `Agda`. Hence we use a trick.

```
|
|plus2 (n::Nat)::Nat = suc (suc n)
|three::Nat = plus2 one
|
|foo::Nat = {}0
|
|-EEE:---F1  Proof: Intro.agda       (Agda:run Ind)--L1--All-------
|Close to: Position "c:/home/program/Agda/Intro.agda" 13 11
|?0 :: Nat
|-EEE:**-F1  Emacs: * Agda version *    (Agda:run Ind)--L1--All----
| marking goals: Intro.agda... done
```

Figure 4: Computation

Append the following line at the tail of "`Intro.agda`" and make `Agda` type-check the program again:

```
foo::Nat = ?
```

Then the character '`?`' must change to the string '`{}0`' (Figure 4). This string is called a **goal** which is a place we can make a code interactively (See Section 5).

Place the cursor on the goal (on the character '`}`' precisely) and invoke **Compute to depth 100** command[8] by `C-c C-x +`. Then we can see a prompt in the minibuffer:

```
|-EEE:**-F1  Emacs: * Agda version *    (Agda:run Ind)--L1--All----
| expression:
```

Give this prompt the expression "`plus2 one`" (and `RET`) and we obtain the result in the subwindow:

```
|
|-EEE:---F1  Proof: Intro.agda       (Agda:run Ind)--L1--All-------
|suc@_ (suc@_ (suc@_ zer@_))
|
|-EEE:**-F1  Emacs: * Agda version *    (Agda:run Ind)--L1--All----
|
```

The expressions `suc@_` and `zer@_` are the most reduced expressions of `suc` and `zer` respectively. The meaning of '`@_`' will be explained in Section 3.11, and now we can ignore it. Thus the result

```
suc@_ (suc@_ (suc@_ zer@_))
```

can be understood as

```
suc (suc (suc zer)).
```

That is the expression we have inferred.

[8] If we can operate with mouse, we can invoke the command by right-clicking on the goal and selecting **Compute to depth 100**.

Type inferences Let us introduce another feature of `Agda`: displaying types of expressions. `Agda` does not have any interface to do it, and thus we use a trick same as computations. Again place the cursor on the goal and invoke **Infer type** by `C-c :`. Similar to the case of computation, the prompt

```
expression:
```

appears. We give it the string "`one`" with `RET` and we can see its type in the subwindow:

```
|
|-EEE:---F1  Proof: Intro.agda       (Agda:run Ind)--L1--All-------
|Nat
|
|-EEE:**-F1  Emacs: * Agda version *    (Agda:run Ind)--L1--All----
|
```

Again invoke **Infer type** on the goal and give the string "`plus2`" to the prompt. We can see `plus2` is of a function type:

```
|
|-EEE:---F1  Proof: Intro.agda       (Agda:run Ind)--L1--All-------
|(n::Nat) -> Nat
|
|-EEE:**-F1  Emacs: * Agda version *    (Agda:run Ind)--L1--All----
|
```

3.2.5 Definitions

A definition binds an `Agda` expression to an identifier. The simplest definition has the following syntax:

$\langle Id \rangle$ `::` $\langle Type \rangle$ `=` $\langle Expr \rangle$

which means that the identifier $\langle Id \rangle$ is bound to the expression $\langle Expr \rangle$.

Notice that the order of definitions is important in `Agda` programs. A definition in a program has, as its context, all constants defined in the earlier part[9] of the program. Therefore, a constants appearing in a definition must have been defined at that time.

Remark. In case $\langle Type \rangle$ is inferred from $\langle Expr \rangle$, the type notation can be omitted.

```
zer' = zer
```

As the right hand side is typed to `Nat`, the left hand side is of type `Nat`, too.

Other types of definitions are introduced in Section 4.

[9] Except for mutual recursive definitions (See Section 3.4.3), and except for locally defined constants (See Section 3.7).

3.2.6 Syntax

The syntactical structure for a function definition is as follows:

$\langle Id \rangle$ $\langle Args \rangle$ `::` $\langle Type \rangle$ `=`$\langle Expr \rangle$

where $\langle Id \rangle$ is the identifier, $\langle Type \rangle$ is the type of the codomain, $\langle Expr \rangle$ is the function body, and $\langle Args \rangle$ is the arguments:

`(`$\langle V1 \rangle$ `::` $\langle T1 \rangle$`)` `(`$\langle V2 \rangle$ `::` $\langle T2 \rangle$`)` $\ldots$ `(`$\langle Vk \rangle$ `::` $\langle Tk \rangle$`)`

The syntactical structure for a function application is as follows:

$\langle Id \rangle$ $\langle Exp1 \rangle$ $\langle Exp2 \rangle$ $\ldots$ $\langle Exp\,k \rangle$

where $\langle Id \rangle$ is the identifier of a function, and each $\langle Exp\,i \rangle$ is an expression.

3.2.7 Exercises

1. Make a function with two arguments: it receives two expression of `Nat`, and returns the second argument.

2. Make a function named '`emb`' representing an embedding of `Nat` into `Int`:

$$\texttt{Nat} \ni x \mapsto \texttt{I}\ x\ \texttt{zer} \in \texttt{Int}$$

3.3 Case expressions

Case expressions are one of basic tools for making functions. It makes possible that a function returns an expression according to the structure of an argument.

Summary

- Examples.
- Meanings of indentations in `Agda`
- Typechecking.
- Syntax

3.3.1 Examples

The first example is a function to flip boolean values. A case expression appears in the right hand side.

```
flip (x::Bool')::Bool' =
        case x of
        (true )-> false
        (false)-> true
```

The meaning of the case expression in the above example is clear. If the argument `x` is the object `true`, then the expression `flip x` is reduced to `false`, and if the argument `x` is `false`, then `flip x` is reduced to `true`.

It is necessary and sufficient for the above case expression to have two branches, since an object of type `Bool'` is either `true` or `false`. We emphasize it is necessary. Unlike some other programming languages, partial functions are not allowed in `Agda`.

Similar to the first example, a function from `Nat` is well-defined if it gives the values for `zer` and for `suc m` where `m::Nat`. The following function checks whether the argument is `zer` or not.

```
isZer (n::Nat)::Bool' =
        case n of
        (zer   )-> true
        (suc n')-> false
```

3.3.2 Indentation

To gain the readability of an `Agda` program, indentations are important. `Agda` regards indentations as syntax, presenting structures of a program. Rules for indentations are intuitive, and are similar to those of Haskell.

The following expression, introduced in the previous subsection,

```
case x of
    (true )-> false
    (false)-> true
```

is with indentations. It indicates two case branches "`(true )-> false`" and "`(false)-> true`" and "`case x of`" and followed by them form a `case` clause. We can write the same expression without indentations as follows:

```
case x of{(true )-> false;(false)-> true}
```

Internally a program using indentations is transformed to one not using indentation rules. Let us see the rules to transform.

```
case x of
    (⟨C1⟩) -> ⟨Exp1⟩
    (⟨C2⟩) -> ⟨Exp2⟩
    (⟨C3⟩) -> ⟨Exp3⟩
  ⟨ODef⟩
```

First `Agda` finds a reserved word '`of`' and absence of '`{`' following it. At the next line `Agda` memorize the indentation and inserts '`{`' at the head of the line. Next the indentation of the line '`(`⟨*C2*⟩`) ->` ' is compared with the memorized indentation. As they are equal, that line belongs to a clause to which the previous line belongs and the separator '`;`' is inserted at the head of that line. The fourth line is processed in the same manner. Last, the indentation of the fifth line is smaller than the memorized indentation. Thus `Agda` inserts '`}`' at the head of the line and discards the memorized indentation. The result is:

```
case x of
    {(⟨C1⟩) -> ⟨Exp1⟩
    ;(⟨C2⟩) -> ⟨Exp2⟩
    ;(⟨C3⟩) -> ⟨Exp3⟩
  }⟨ODef⟩
```

There are some expressions together with clauses: record types and their objects (Section 3.6), mutual recursive definitions (Section 3.4.3), let expressions (Section 3.7), package definitions (Section 4), and case expressions.

3.3.3 Typecheck and Computations

Let us see that the codes defined here are accepted by `Agda`. Input the definitions of `flip` and `isZer` at the tail of "`Intro.agda`", and make `Agda` load the whole program.

Next, let us compute some expressions. For example

```
flip true
isZer zero
isZer three
```

To do it:

1. Append the line "`foo'::?  = ?`" at the end of "`Intro.agda`",
2. Make `Agda` load the program again,
3. Place the cursor on a goal, and
4. Invoke **Compute to depth 100** command.

```
|
|isZer (n::Nat)::Bool' =
|    case n of
|    (zer   )-> true
|    (suc n')-> false
|
|foo':: ? = ?
|
```

3.3.4 Syntax

Let T be a type. A case expression for an expression of type T is written in the following form:

case $\langle expr \rangle$ of
  ($\langle Cons1 \rangle$) -> $\langle Exp\ 1 \rangle$
  ($\langle Cons2 \rangle$) -> $\langle Exp\ 2 \rangle$
    ...
  ($\langle Consk \rangle$) -> $\langle Exp\ k \rangle$

or equivalently

case $\langle expr \rangle$ of{($\langle Cons1 \rangle$) -> $\langle Exp\ 1 \rangle$; ($\langle Cons2 \rangle$) -> $\langle Exp\ 2 \rangle$; ...;
    ($\langle Consk \rangle$) -> $\langle Exp\ k \rangle$}

Here $\langle expr \rangle$ is an expression of type T, each $\langle Cons\ i \rangle$, is a constructor (maybe with arguments) of T, and $\langle Exp\ i \rangle$ is a function body. Each constructor (with arguments) of type T must correspond to only one of $\langle Cons\ i \rangle$, and all $\langle Exp\ i \rangle$'s must be of the same type.

Remark. If ⟨*expr*⟩ is too complicated to infer its type, then Agda says error. We can avoid this problem by means of local definition (Local definitions are introduced in Section 3.7). Concretely, if the following expression

```
bar :: Nat =
    case ComplicatedExpressionOfNat of
        (zer  )-> ...
        (suc m)-> ...
```

is not accepted because of the expression *ComplicatedExpressionOfNat*, we can modify that expression into

```
bar :: Nat =
    let n::Nat = ComplicatedExpressionOfNat
    in
        case n of
            (zer  )-> ...
            (suc m)-> ...
```

which will be accepted.

3.3.5 Exercises

1. Make a function named `neg` defined by

$$\texttt{Int} \ni (\texttt{I}\ x\ y) \to (\texttt{I}\ y\ x) \in \texttt{Int}.$$

2. Define `IfNat` function. Its specification is as follows:
    - It receives three arguments: `b :: Bool'`, `cT :: Nat`, and `cF :: Nat`.
    - If the first argument `b` is `true`, then `IfNat` returns `cT`.
    - If the first argument `b` is `false`, then `IfNat` returns `cF`.

3.4 Recursive functions

Recursive functions are naturally represented by case expressions.

Summary

- Examples.
- Termination check
- Mutual recursive functions
- Infix operators

3.4.1 Examples

Let us define an addition function on `Nat`.

```
add (m::Nat) (n::Nat)::Nat
    = case m of
      (zer   )-> n
      (suc m')-> suc (add m' n)
```

In the definition of the function `add` there exists `add` itself in the right hand side, thus this definition is recursive.

That definition may seem strange. Though the definition of `add` does not complete, that constant has an occurrence in the right hand side. However it is legal. Typechecking only judges whether a given expression is well-typed or not, and the constant `add` have been typed to[10] `(m::Nat)->(n::Nat)->Nat`. When computing an expression in which `add` occurs, its definition is needed.

Similarly, a multiplication function on `Nat` is defined:

```
mul (m::Nat) (n::Nat)::Nat
    = case m of
      (zer   )-> zer
      (suc m')-> add n (mul m' n)
```

Exercise. Input the examples above at the tail of "`Intro.agda`" and typecheck it. Moreover compute some expressions like as

```
add three one
mul three three
```

3.4.2 Termination check

The authors apologize readers that Termination check does not work well on `Agda` *version we are using. We remark that contents here are checked on other versions.*

Consider the following function

```
nonT (m::Nat)::Nat
    = nonT (suc m)
```

It is a legal definition, indeed typechecking it succeeds (Try it!). But clearly we cannot get a return value with this function: to compute `nonT` m we need the value of `nonT` $(m+1)$, and to compute the value of `nonT` $(m+1)$ we need the value of `nonT` $(m+2)$, and to compute the value of `nonT` $(m+2)$, ..., the computation does not finish.

To check whether a computation of an expression in a code terminates or not, **Check termination** command is prepared. Input the function definition of `nonT` and invoke Check termination command by `C-c C-x C-t`. Then the result is displayed in the sub window.

```
|
|-EEE:---F1  Proof: Intro.agda      (Agda:run Ind)--L1--All--------
|At: "Intro.agda", line 20, column 0
```

[10] Notations like as `(m::Nat) -> Nat` is introduced in Section 3.2, 3.5.

```
|The call: nonT
|might lead to non-termination
|-EEE:**-F1  Emacs: * Agda version *    (Agda:run Ind)--L1--All----
|
```

If we delete or comment out the definition of `nonT`, input the definition of `add`, and invoke Check termination command, then `Agda` tells us that all expressions in the code **terminates**.

3.4.3 Mutual recursive definitions

Consider the following two functions f and g that depend on each other:

$$f(0) = 1, g(0) = 0$$
$$f(n+1) = 3f(n) + g(n)$$
$$g(n+1) = f(n) + 3g(n)$$

Try implementing these functions in `Agda`. How can we do it?

In `Agda`, any constant in a program must be well-typed or bound to an expression till it occurs in another expression. But in this case, each of the functions f and g refers to each other in their definitions. To define such functions, we need to put definitions in a `mutual` clause:

```
mutual
  f (n::Nat):: Nat =
    case n of
    (zer  )-> one
    (suc n')-> add (mul three (f n')) (g n)
  g (n::Nat):: Nat =
    case n of
    (zer  )-> zero
    (suc n')-> add (f n') (mul three (g n'))
```

It is equivalently written as:

```
mutual{
f (n::Nat):: Nat =
    case n of
    (zer  )-> one
    (suc n')-> add (mul three (f n')) (g n);
g (n::Nat):: Nat =
    case n of
    (zer  )-> zero
    (suc n')-> add (f n') (mul three (g n'))
}
```

3.4.4 Operators

Let us define an equality of `Nat`. It is realized as a function returning an object in `Bool'` for given two expressions of type `Nat`.

```
(==) (m::Nat) (n::Nat)::Bool'
    = case m of
```

```
(zer    )->
  case n of
  (zer    )-> true
  (suc n')-> false
(suc m')->
  case n of
  (zer    )-> false
  (suc n')-> m' == n'
```

The identifier of this function is the string "`==`". Such identifiers that consist of symbolic characters are called **Operators**. An operator must be referred with parenthesis (see the left hand side in the example above) except for one case. When an operator is bound to a two variable function, it is dealt with as an infix operator. (See the last line in the example above.)

Remark. `Agda` does not separate symbols adjacent to each other; those symbols are recognized as an operator (a lambda expression is used in the following example. See Section 3.5):

```
plus2::Nat -> Nat =\(x::Nat) -> suc (suc x)
```

`Agda` regards the string '`=\`' as a single operator and says error. We must place a whitespace between '`=`' and '`\`'.

For later use, let us define two operators:

```
(+) (m::Nat) (n::Nat)::Nat = add m n
(*) (m::Nat) (n::Nat)::Nat = mul m n
```

3.4.5 Exercises

1. Define an equality function '`(===)`' on `Int`. It is defined by the rule

$$(m, n) \equiv (m', n') \Leftrightarrow m + n' = n + m'$$

   where the equality and addition in the right hand side are on `Nat`.

2. Define a function `even` from `Nat` to `Bool'` returning `true` for even numbers.

3. We have defined `ListNat` and we write its object like as[11]

   ```
   conN zer (conN zer (conN (suc zer) nilN))
   ```

   We want to write it in simpler way like as

   ```
   zer :. (zer :. ((suc zer) :. nilN))
   ```

   Define an infix operator '`:.`' which constructs `ListNat`.

4. Some functional programming languages have `head` functions. It receives a list and returns an element at the head of the list. Try defining `headNat` function, which works like as `head`, for `ListNat`. Is it possible? Why can/cannot we do it?

[11] It is presented as [0,0,1] in Haskell.

3.5 Functions(lambda expressions)

The second way (the first way is introduced in Section 3.2) to define functions is to use lambda expressions which represents abstractions.

3.5.1 Examples

The following is a translation of the function definition of `flip`:

```
flip'::Bool' -> Bool'
    = \(x::Bool') ->
        case x of
        (true )-> false
        (false)-> true
```

Similarly the following code is a translation of `add`:

```
add'::Nat->Nat->Nat
    = \(m::Nat)->
      \(n::Nat)->
    case m of
      (zer   )-> n
      (suc m')-> suc (add' m' n)
```

Owing to lambda expressions, even functions are dealt with as data:

```
twice::(Nat->Nat)->Nat->Nat =
    \(f::Nat->Nat)->
    \(x::Nat)->
    f (f x)
```

This function receives a function f and an object x of `Nat` and returns the value $f(f(x))$.

3.5.2 Syntax

A function type is described with ‘ `->` ’: to describe a function type from ⟨*Type1*⟩ to ⟨*Type2*⟩, the following expression is used:

⟨*Type1*⟩ `->` ⟨*Type2*⟩

An object of a function type comes with a lambda expression:

`\(`⟨*Var*⟩ `::` ⟨*Type*⟩`)` `->` ⟨*Expr*⟩

where ⟨*Var*⟩ is a variable of type ⟨*Type*⟩ and ⟨*Expr*⟩ is an `Agda` expression.

Of course ⟨*Type*⟩ in the definition

⟨*Id*⟩ `::` ⟨*Type1*⟩ `->` ⟨*Type2*⟩ `=\(`⟨*Var*⟩ `::` ⟨*Type*⟩`)` `->` ⟨*Expr*⟩

must coincide with ⟨*Type1*⟩.

3.5.3 Exercises

1. Make `mapNat` function. It takes

   a function f `::` `Nat -> Nat` and
   a list x_1`:.(` x_2`:.(`$\ldots$`:.(`x_n`:. nilN)))` of `ListNat`

   as arguments, and returns

   a list $f(x_1)$`:.(` $f(x_2)$`:.(`$\ldots$`:.(`$f(x_n)$`:. nilN)))` of `ListNat`

   Or `mapNat` returns '`nilN`' if the list in arguments is '`nilN`'.

3.6 Record types

3.6.1 Examples

Records are unordered labeled products. A record type is expressed by a clause following '`sig`' ('`sig`' for 'signature'):

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

The type `Tuple` consists of two types: `Bool'` and `Nat`. `Set` can be ignored now. It is explained in Section 3.9. An object of `Tuple` has two members and is defined as a clause following '`struct`':

```
aPair::Tuple = struct
                fst::Bool' = true
                snd::Nat = suc zer
```

To access a member of `aPair`, a dot notation is used like as

```
n::Nat = aPair.snd
```

3.6.2 Syntax

A record type is defined as follows:

`sig`
    ⟨*Id1*⟩ `::` ⟨*Type1*⟩
    ⟨*Id2*⟩ `::` ⟨*Type2*⟩
        . . .
    ⟨*Idk*⟩ `::` ⟨*Typek*⟩

Or equivalently

`sig{` ⟨*Id1*⟩ `::` ⟨*Type1*⟩`;` ⟨*Id2*⟩ `::` ⟨*Type2*⟩`;` . . . `;` ⟨*Idk*⟩ `::` ⟨*Typek*⟩ `}`

Each ⟨*Id i*⟩ [resp. ⟨*Type i*⟩] is the identifier [resp. the type] of a member in the record type.

A `struct` clause defines an object of a record type:

```
struct
    ⟨Id1⟩ :: ⟨Type1⟩ = ⟨Expr1⟩
    ⟨Id2⟩ :: ⟨Type2⟩ = ⟨Expr2⟩
          ...
    ⟨Idk⟩ :: ⟨Typek⟩ = ⟨Exprk⟩
```

Or

```
struct{ ⟨Id1⟩ :: ⟨Type1⟩ = ⟨Expr1⟩; ⟨Id2⟩ :: ⟨Type2⟩ = ⟨Expr2⟩; ... ;
        ⟨Idk⟩ :: ⟨Typek⟩ = ⟨Exprk⟩ }
```

Notice that a record type corresponding to the `struct` clause must have been defined before typechecking it.

3.7 Local definitions

3.7.1 Examples

A `let` expression make it possible to use local bindings.

```
aNat::Nat = let
                n::Nat = add three (add three one)
            in
                mul n n
```

The constant `aNat` equals to the following expression

```
mul (add three (add three one)) (add three (add three one))
```

Due to the local constant `n` we do not have to write repeated complicated expression, and `n` is not bound to `add three (add three one)` outside `let`-clause and the expression following `in`.

Moreover `let` expressions make typchecking effective. In the above example, the expression `mul n n` following `in` is typechecked if `mul` and `n` are typechecked, and `n` has been already typed to `Nat` in `let` clause. However to typecheck

```
mul (add three (add three one)) (add three (add three one))
```

`(add three (add three one))` are typechecked twice. In Section 3.3.4 the other benefit of `let` expressions are introduced.

3.7.2 Syntax

A `let` expression consists of `let` clause and an expression following the reserved word `in`.

```
let
    ⟨Def1⟩
    ⟨Def2⟩
      ...
    ⟨Defk⟩
in
    ⟨Expr⟩
```

Or equivalently

```
let{ ⟨Def1⟩; ⟨Def2⟩;  ...; ⟨Defk⟩ } in { ⟨Expr⟩ }
```

Each ⟨*Def i*⟩ is a definition: it has a form

```
⟨Id i⟩ :: ⟨Type i⟩ = ⟨Expr i⟩
```

or

```
⟨Id i⟩ :: ⟨Type i⟩
⟨Id i⟩ = ⟨Expr i⟩
```

and ⟨*Expr*⟩ is an expression.

Local definitions in `let` clause are order-sensitive. One definition may depend on previous definitions in that clause. For instance it is legal to write

```
let
    c = a
    d = c
in
    e
```

but not to write

```
let
    d = c
    c = a
in
    e
```

3.7.3 Exercises

This is an exercise for the previous subsections. For integers m and n where $n > 0$, there exists integers q, r such that

$$m = qn + r$$

and that $0 \leq r < n$. Make a function calculating q and r. (Hint: (1) the return value must be a pair of integers; (2) we can do this exercise after reading the following section, especially Section 5.)

The following are some test cases. Does the function we made return expected values for them?

m	n
5	3
-4	3
0	2
4	0

3.8 Short summary

In the previous subsections, some basic expressions are introduced:

- Data types: data type definitions and record types
- Function types

- Function definitions
- Case expressions
- Let expressions

They let us deal with Simple type theory in `Agda`. We can describe types independent of other types/objects as well as their objects.

In the rest of this section, advanced features are introduced, especially dependent types. In Dependent type theory types and objects are even. A type must be typed as well as an object and a type may be dependent on other types and objects.

By introducing Dependent type theory, the most general descriptions of expressions introduced in the previous subsections are shown.

3.9 Types

In Dependent type theory, a type is not distinguished from other objects. Thus a type must be typed to an other type, and a framework for that is supplied.

3.9.1 Examples

At the first, Let us see a special type `Set`. All types that have been defined in this tutorial are typed to `Set`:

```
Bool'::Set
ListNat::Set
Int::Set
```

On an arbitrary type X, lists with elements in X are important data types. To define them, we should pass X as an argument, that is the following data type depends on an object X of `Set`.

```
data List' (X::Set) = nil | con (x::X) (xs::List' X)
```

Consider a generalized `length` function. The type of the second argument depends on the first argument:

```
length (X::Set) (xs::List' X)::Nat =
    case xs of
        (nil      )-> zer
        (con x xs')-> suc (length X xs')
```

3.9.2 Typing rules

Small types There is a built-in expression `Set` in `Agda`, and it works as a 'type of types'. An object of `Set` is a type, which corresponds to a 'set' in the ordinary mathematics, also known as a 'small type.'

- A data type is an object of `Set`.
- A `->` B is an object of `Set`, provided that A and B are objects of `Set`. It is a function type.

- `(`x `::` A`) ->` 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. Function types introduced above are specialized dependent function types: the type B is not dependent on A.

At last small types are closed under dependent function type formation, meaning that when A and B are sets, `(`x `::` A`) ->` B is also a set.

Large types Here one question occurs in our mind. How about `Set`? Every legal expression in `Agda` code must be of a legal type, and `Set` is a legal expression in `Agda`. Hence `Set` must be typed, but it must not be typed to `Set` itself: that would lead to paradoxes.

Types in `Agda` are stratified by their "size": the type `Set` and all objects of `Set` (with the type forming operation '`->`') are typed to `Type`, a built-in type, and similarly the type `Type` and all objects of it are typed to a sort '`#2`', and so on[12]. `Set`, `Type`, are called 'sorts'.

3.10 General expressions of functions

As shown in Section 3.9, a function type is expressed like as `(`x `::` A`) ->` B, in which the formal parameter x appearing in B must be shown explicitly. Thus a one-variable function is generally defined[13] as follows:

⟨*Id*⟩ `::` `(`⟨*V1*⟩ `::` ⟨*T1*⟩`) ->` ⟨*Type*⟩ `=` `\(`⟨*V'1*⟩ `::` ⟨*T1*⟩`) ->` ⟨*Expr*⟩

where ⟨*Id*⟩ is the identifier, ⟨*V1*⟩ and ⟨*V'1*⟩ are variables of type ⟨*T1*⟩. Notice that ⟨*Type*⟩ may depend on ⟨*V1*⟩.

In the above definition, by considering ⟨*V1*⟩ `::` ⟨*T1*⟩ as an assumption for the constant ⟨*Id*⟩ we have a variant of the definition:

⟨*Id*⟩ `(`⟨*V1*⟩ `::` ⟨*T1*⟩`)` `::` ⟨*Type*⟩ `=`⟨*Expr*⟩

introduced in Section 3.2.

Moreover, when ⟨*Type*⟩ does not depend on ⟨*V1*⟩, we have another variant

⟨*Id*⟩ `::` ⟨*T1*⟩ `->` ⟨*Type*⟩ `=\(`⟨*V'1*⟩ `::` ⟨*T1*⟩`) ->` ⟨*Expr*⟩

introduced in Section 3.5.

3.11 Data type definitions

3.11.1 Examples and description

In Section 3.1 we have defined `Nat` by the following line:

```
data Nat = zer | suc (m::Nat)
```

It is a syntax sugar of the following lines:

```
Nat::Set = data zer | suc (m::Nat)
zer::Nat = zer@_
suc (m::Nat)::Nat = suc@_ m
```

[12]Only `Set` and `Type` appear in this tutorial. It is sufficient.

[13] We can explain the cases of multi-variable functions similarly. They are the cases ⟨*Type*⟩ in the code is also a function type.

The first line defines the data type `Nat` and the following two lines define functions. What is '`@_`'?

In precise, a constructor must come with its type annotation in `Agda`. Owing to it, plural data types can have constructors with a common identifier. For example, the occurrences of the following constructors in a program does not make any troubles:

`true@Bool'`
`true@Bool` (`Bool` is a built-in type)
`suc@Nat m` (where `m::Nat`)
`suc@Nat zer@Nat`

In case a constructor's type is uniquely inferred, we can replace its annotation to the character '`_`'. For example, when there is no type, which has a constructor named "`zer`" and "`suc`" other than `Nat` and `Nat` respectively, we can write as follows:

`true@_`	instead of	`true@Bool'`
`suc@_ zer@_`	instead of	`suc@Nat zer@Nat`

Remark on case expressions Even if we define a data type by the manner

```
List' (X::Set)::Set = data ......,
```

'`@_`' can be omitted in case expression:

```
tail (X::Set):: (xs::List' X) -> List' X =
    \(xs::List' X)->
    case xs of
    (nil      )-> nil@_
    (con x xs')-> xs'
```

That is why the type of `xs` is trivially inferred and type annotations for `nil` and `con` are not needed.

3.11.2 Syntax

Data type definition is simply an `Agda` expression. Although it can be a subexpression of other expressions, it is recommended that we deal with a data type definition as the right hand side value.

`data` $\langle \mathit{Cons\ 1} \rangle$ `|` $\langle \mathit{Cons\ 2} \rangle$ `|` ... `|` $\langle \mathit{Cons\ k} \rangle$

Each $\langle \mathit{Cons\ i} \rangle$ can have arguments as introduced in Section 3.1.

A constructor is referred to by the form

$\langle \mathit{Id} \rangle$`@`$\langle \mathit{Type} \rangle$

where $\langle \mathit{Id} \rangle$ is an identifier of the constructor and $\langle \mathit{Type} \rangle$ is its type.

3.11.3 Exercises

1. Input the following definition in the file "`Intro.agda`" and typecheck:

   ```
   Sgn::Set = data pos | zer | neg
   ```

   Does any error occur?

3.12 Inductive data definitions

Inductive data type definitions make it possible to define types with indices.

Let us see an example. Let A be a type and consider the type of lists of A with a fixed length, denoted by

$\texttt{Vec}\ (A :: \texttt{Set})\ (n :: \texttt{Nat})$

Its objects are constructed with respect to the argument n:

- `Zero` is the unique object of $\texttt{Vec}\ A\ 0$,
- For $n :: \texttt{Nat}$, all pairs of $a :: A$ and $v :: \texttt{Vec}\ A\ n$ are objects of $\texttt{Vec}\ A\ (n+1)$ (and every object of the type is a pair described above).

We cannot define $\texttt{Vec}(A :: \texttt{Set})(n :: \texttt{Nat})$ by one `data` expression, since distinct constructors are needed depending on the argument n.

An `idata` expression makes us possible to define $\texttt{Vec}\ A\ n$ in this way:

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

It is interpreted as

- For $A :: \texttt{Set}$, $\{\texttt{Vec A n}\}_{\texttt{n::Nat}}$ is a family of types with indices in `Nat`.
- An expression `Zero@_` is an object of `Vec A zer@_`.
- An expression `Cons@_ n a v` is an object of `Vec A (suc@_ n)`.

Each expression following "`:: _`" is an index to each data type.

Remark. Like as an underscore '`_`' in the expression '`true@_`', the underscore following '`::`' in the definition of `Vec A zer@_`:

```
Zero :: _ zer@_
```

is a replacement for '`Vec A`'. This expression can be trivially inferred, and is omitted. But '`:: _`' is a syntax. We can not change it to '`Zero :: Vec A zer@_`'.

3.12.1 Syntax

An `idata` expression restricts the type of the left hand side: it must be a function type.

$\langle \mathit{Id} \rangle :: \langle \mathit{FType} \rangle$ `= idata` $\langle \mathit{ICon}\ 1 \rangle \,|\, \langle \mathit{ICon}\ 2 \rangle \,|\, \ldots \,|\, \langle \mathit{ICon}\ k \rangle$

where $\langle \mathit{Id} \rangle$ is the identifier, $\langle \mathit{FType} \rangle$ is a function type with its codomain `Set` (or `Type`). Moreover $\langle \mathit{ICon}\ i \rangle$ has a form

$\langle \mathit{CId} \rangle\ \langle \mathit{Args} \rangle :: \_\ \langle \mathit{Idx} \rangle$

where $\langle \mathit{CId} \rangle$ is the identifier of the constructor, $\langle \mathit{Args} \rangle$ is an argument list (same as ordinary data type definitions) and $\langle \mathit{Idx} \rangle$ is an expression whose type is the domain of $\langle \mathit{FType} \rangle$.

Note that it is not necessary to define a family of types for all indices. The following `idata` definition works without problems:

```
Vec (A::Set) :: Nat->Set
    = idata Zero :: _ zer@_
```

As a result, only `Vec A zer@_` is defined.

When a family of types has more than one indices, $\langle Idx \rangle$ is a sequence of expressions. For example let us define a family of types of limited natural numbers: for given natural numbers m and n, x is an object of `LimitedNat` $m\ n$ if and only if $m \leq x \leq m + n$. They are defined by means of `idata` expression:

```
LimitedNat :: Nat -> Nat -> Set =
    idata Lb (m,n::Nat) ::_ m n |
          Suc (m,n::Nat) (x::LimitedNat m n) ::_ m (suc@_ n)
```

3.13 Hidden arguments

A hidden argument (also referred to an implicit argument) are a newer feature of `Agda`. With that feature, we do not have to write arguments which are trivially inferred.

3.13.1 Examples

In dealing with an expression of `Agda`, all variables appearing in it must have been declared. Let us see the following example (`List'` is also defined in Section 3.9):

```
List' (X::Set) :: Set
    = data nil | con (x::X) (xs::List' X)

nil (X::Set) :: List' X
    = nil@_
con (X::Set)(x::X)(xs::List' X) :: List' X
    = con@_ x xs
```

The first argument of `con` function is necessary to indicate the type of elements in lists, though they do not occur in the right hand sides. In other words, that argument is needed in the definition only, but is not needed in applications of `con`. In fact, in the line below, we can infer that X is `Nat` since `zero` and `nil Nat` are typed to `Nat` and `List' Nat` respectively.

```
aList::List' Nat = con X zero (nil Nat)
```

With hidden arguments, we can modify definitions of `nil`, and `con`:

```
nil' (X::Set) :: List' X
    = nil'@_
con' (|X::Set)(x::X)(xs::List' X) :: List' X
    = con@_ x xs
```

Notice that the difference is the symbol '`|`' (a vertical bar) followed by the first argument `X`. Now we can omit the first argument, that is

```
aList::List' Nat = con' zero (nil' Nat)
```

is a legal definition[14] typed to `List Nat`.

[14] Moreover `con' zero nil'` is well-typed if we modify the definition of `nil'` to

3.13.2 Syntax

By placing a vertical bar '`|`' in a definition we can make an argument hidden. There are some cases.

- Putting on '`|`' at the left of a formal variable to be hidden:

  ⟨*Id*⟩ `(|`⟨*HidVar*⟩ `::` ⟨*T1*⟩`)` `::` ⟨*Type*⟩ `=` ⟨*Body*⟩

- Putting on the right of a type to be hidden:

  ⟨*Id*⟩ `::` `(`⟨*Var*⟩ `::` ⟨*Type*⟩`)` `|->` ⟨*Type'*⟩ `=` ⟨*Body*⟩

- In a lambda expression, putting on the right of a formal variable to be hidden:

  `\(`⟨*Var*⟩ `::` ⟨*Type*⟩`)` `|->` ⟨*Expr*⟩

When we want to give an expression explicitly to a hidden argument, we can simply put the symbol '`|`' on the left of arguments. For example, the following is a legal expression with definitions introduced above:

```
con |Nat zero nil
```

Remark. If it is unsure about hidden arguments we had better give arguments explicitly like as in the last remark.

3.14 Example program

Here we show a complete program consisting of codes appearing in the present section.

```
----  Intro.agda  ----
-- Section 3.1
data Bool' = true | false
data Nat = zer | suc (m::Nat)

-- Section 3.2
zero::Nat = zer
one::Nat = suc zer

succ (m::Nat)::Nat = suc m
plus2 (n::Nat) ::Nat = suc (suc n)
three::Nat = plus2 one

foo::Nat = ?

-- Section 3.3
flip (x::Bool')::Bool' =
    case x of
    (true )-> false
    (false)-> true
```

```
nil' (|X::Set)::List' X = nil'@_
```

```
isZer (n::Nat)::Bool' =
    case n of
    (zer   )-> true
    (suc n')-> false

foo':: ? = ?

-- Section 3.4
add (m::Nat) (n::Nat)::Nat =
    case m of
        (zer   )-> n
        (suc m')-> suc (add m' n)

mul (m::Nat) (n::Nat)::Nat =
    case m of
        (zer   )-> zer
        (suc m')-> add n (mul m' n)

foo''::? = ?

nonT (m::Nat)::Nat
    = nonT (suc m)

foo'''::? = ?

mutual
    f (n::Nat)::Nat =
        case n of
        (zer   )-> one
        (suc n')-> add (mul three (f n')) (g n)
    g (n::Nat)::Nat =
        case n of
        (zer   )-> zero
        (suc n')-> add (f n') (mul three (g n'))

(==) (m::Nat) (n::Nat)::Bool' =
    case m of
        (zer   )->
            case n of
                (zer   )-> true
                (suc n')-> false
        (suc m')->
            case n of
                (zer   )-> false
                (suc n')-> m' == n'

(+) (m::Nat) (n::Nat)::Nat = add m n
(*) (m::Nat) (n::Nat)::Nat = mul m n

-- Section 3.5
flip'::Bool'->Bool' =
    \(x::Bool')->
        case x of
        (true )-> false
```

```
        (false)-> true

add'::Nat->Nat->Nat =
    \(m::Nat)->
    \(n::Nat)->
        case m of
        (zer   )-> n
        (suc m')-> suc (add' m' n)

twice::(Nat->Nat)->Nat->Nat =
     \(f::Nat -> Nat)->
     \(x::Nat)->
     f (f x)

foo''''::? = ?


-- Section 3.6
Tuple::Set = sig
            fst::Bool'
            snd::Nat

aPair::Tuple = struct
                fst::Bool' = true
                snd::Nat = suc zer

n::Nat = aPair.snd

-- Section 3.7
aNat :: Nat =
    let
        n::Nat = add three (add three one)
    in
        mul n n

-- Section 3.9
data List' (X::Set) = nil | con (x::X) (xs::List' X)

length (X::Set) (xs::List' X)::Nat =
    case xs of
        (nil      )-> zer
        (con x xs')-> suc (length X xs')

-- Section 3.11
tail (X::Set)::(xs::List' X)-> List' X =
     \(xs::List' X)->
     case xs of
     (nil      )->
       nil@_
     (con x xs')->
       xs'

-- Section 3.12
Vec (A::Set) :: Nat->Set
```

```
    = idata Zero :: _ zer@_ |
            Cons (n::Nat) (a::A) (v::Vec A n) :: _ (suc@_ n)

LimitedNat :: Nat->Nat->Set =
    idata Lb (m,n::Nat)::_ m n |
          Suc (m,n::Nat) (x::LimitedNat m n) ::_ m (suc@_ n)

-- Section 3.13
nil' (X::Set)::List' X =
    nil@_
con' (|X::Set)(x::X)(xs::List' X)::List' X =
    con@_ x xs
```

4 Packages

There is a package system in `Agda`. A package is series of definitions, being allowed passing arguments from outside. With packages, we do not have to write the same definitions in different programs. Moreover, we can deal with a constant, which is typed but not defined, appearing in a package. This feature makes the package more abstract. In particular it is very beneficial to implementing logic. An axiom of a logic is realized with that feature (See Section 6).

Here we make a package including definitions introduced in Section 3. That package will be used in the following sections.

Summary

- Standard packages as examples.
- How to use packages. Configurations.
- How to define packages.

4.1 Reviewing standard packages

First, we review and use standard library packages. They are not different from other `Agda` programs, and hence they will give us good examples of not only usages of packages but also programs in `Agda`.

Remark. There is a library manual coming together with library packages. Hence we do not deal with the contents of packages in detail.

4.1.1 Package location

First let us check we can use standard packages without troubles. `Agda` installer must have copied package files into a directory (usually named "`Hedberg`") in our system. If unsure, we should check the existence of packages. It will be sufficient to search the file "`SET.alfa`".

The location of the directory which has package files is preserved in Emacs variable '`agda-include-path`' (Section 4.3.1 describes in detail.). If the variable has the string "`/usr/local/Hedberg/`", then the reference "`SET.alfa`" in this tutorial points to a file "`/usr/local/Hedberg/SET.alfa`".

Is the variable "`agda-include-path`" set correctly? Although there are some ways to check/set the variable, here we use a graphical user interface. Open a new buffer[15] named "`PackagesEg.agda`" to write examples in this section. Invoke `M-x customize-group` (`M-x` followed by the string '`customize-group`') in `agda-mode`, and we can see the buffer to customize an environment for `Agda`.

[15] We must be in `agda-mode` to configure settings with GUI.

```
|This is a customization buffer for group Agda.
|‘Raised’ buttons show active fields; type RET or click mouse-1
|on an active field to invoke its action.  Editing an option value

|Agda Include Path: Show
|   State: hidden, invoke "Show" in the previous line to show.
|List of directories for searching included files
|
```

We can find the item entitled "Agda Include Path" with a button. Clicking the button shows us the contents of `agda-include-path`

```
|Agda Include Path: Hide
|INS DEL Directory: /usr/local/lib/Alfa/Library/Hedberg/
|INS
|   State: this option has been changed outside the customize buffer.
|List of directories for searching included files
```

Now we can set or check the contents of `agda-include-path`: to set it, we can simply edit the string. Modified settings work when we save it: selecting a button at the head of this buffer.

```
|
|Operate on everything in this buffer:
| [Set for Current Session] [Save for Future Sessions]
| [Reset] [Reset to Saved][Reset to Standard]  [Bury Buffer]
|
```

We can leave this buffer. Change the current buffer to "`PackagesEg.agda`".

4.1.2 Directory Tree

Some fundamental packages are placed on the top of directories and the rests are placed appropriate subdirectories:

- Placed in the directory `agda-include-path` points:

  `AlgLaw0.alfa`,
  `AlgLaw1.alfa`,
  `EqLem.alfa`,
  `SET.alfa`, and
  `TYPE.alfa`.

- Placed in the subdirectories of the directory `agda-include-path` points:
  - `Op/*`: Contains packages on operations for each data type defined in `SET`.
  - `Logic/*`: Contains packages on logics.
  - `Datoid/*`: Contains realizations of some data types as datoids. The type `Datoid` is defined in `SET`.
  - `Setoid/*`: Contains realizations of some data types as setoids. The type `Setoid` is defined in `SET`.
  - `Card/*`: Contains packages on the equational theory of setoids.
  - `Nat/*`: Contains packages on the equational theory of natural numbers.

4.1.3 SET.alfa

The most basic package is named SET, which is contained in the file SET.alfa. The package contains the basic inductive data types, related combinators and the logical constants and the basic definitions for equality.

Let us review contents of the package SET; Some lines at the head of "SET.alfa" are printed in Figure 5. Definitions in the package SET are classified to five groups.

Auxiliary: Some combinators and type abbreviations are defined.

Sets: It consists of basic data types, such as the empty set, the singleton set, the dependent pairs, sets for natural numbers, lists and so on together with the arithmetical constants (operating on sets) and the inductive equality proof sets.

Propositions: The proof theoretical interpretation of the arithmetical constants.

Booleans: The universe of truth values in the classical logic.

Equality: Fundamentals of equality. Types of sets together with equivalences are introduced.

```
package SET where
  ----------------------------------------------
  -- Auxiliary.
  ----------------------------------------------
  Unop (X::Set) :: Set
    = X -> X
  Binop (X::Set) :: Set
    = X -> X -> X
  Pred (X::Set) :: Type
    = X -> Set
  Pow = Pred
  Rel (X::Set) :: Type
    = X -> X -> Set
```

Figure 5: The head of SET.alfa

4.1.4 Typical example

Let us review the head of package file Op/Nat.alfa. It includes most of typical usages about packages:

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

package OpNat where
  open SET  use  Nat,  Fin,  Bool
  -- Arithmetical operations.
```

```
zero :: Nat
  = zer@_
unit :: Nat
  = suc@_ zero
succ (x::Nat) :: Nat
  = suc@_ x
pred (x::Nat) :: Nat
  = case x of {
      (zer)    -> zer@_;
      (suc x') -> x';}
(+) (m::Nat)(n::Nat) :: Nat
  = case m of {
      (zer)    -> n;
      (suc m') -> suc@_ ((+) m' n);}
(*) (m::Nat)(n::Nat) :: Nat
  = case m of {
      (zer)    -> zer@_;
      (suc m') -> (+) n ((*) m' n);}
                         .........
```

Although details are explained in the sequel, we can understand the code:

The 1st line: "`--#include`" is a directive to load the contents of the file "`SET.alfa`" (Section 4.3.1)

The 2nd line: The package named "`OpNat`" is defined from here. (Section 4.2)

The 3rd lines: It is declared that constants `Nat`, `Fin`, and `Bool` defined in the package `SET` are used in the latter part of this package. (Section 4.3.2)

The 5th line, the 7th line,. . . : Definitions of constants in the package `OpBool`.

4.2 Package Definition

A package is defined by the following manner:

`package` ⟨*PackId*⟩ ⟨*Args*⟩`where`
  ⟨*Def1*⟩
  ⟨*Def2*⟩
    . . .
  ⟨*Defn*⟩

where ⟨*PackId*⟩ is the identifier of the package, and ⟨*Args*⟩ is an arguments. Of course we can make a package without indentations.

`package` ⟨*PackId*⟩ ⟨*Args*⟩`where{`
  ⟨*Def1*⟩`;`
  ⟨*Def2*⟩`;`
    . . .
  ⟨*Defn*⟩
  `}`

The following is a package named "IntroLib.agda" including the definitions introduced in Section 3.

```
----  IntroLib.agda

package IntroLib where
    data Bool' = true | false
    data Nat = zer | suc (m::Nat)

    add (m::Nat) (n::Nat)::Nat =
        case m of
            (zer   )-> n
            (suc m')-> suc (add m' n)

    mul (m::Nat) (n::Nat)::Nat =
        case m of
            (zer   )-> zer
            (suc m')-> add n (mul m' n)

    (==) (m::Nat) (n::Nat)::Bool' =
        case m of
            (zer   )->
                case n of
                    (zer   )-> true
                    (suc n')-> false
            (suc m')->
                case n of
                    (zer   )-> false
                    (suc n')-> m' == n'

    (+) (m::Nat) (n::Nat)::Nat = add m n
    (*) (m::Nat) (n::Nat)::Nat = mul m n


    data List' (X::Set) = nil | con (x::X) (xs::List' X)

    length (X::Set) (xs::List' X)::Nat =
        case xs of
            (nil      )-> zer
            (con x xs')-> suc (length X xs')

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

4.3 How to use a package

4.3.1 Including a file

To load definitions in an external file, we must place the following line at the head of Agda program:

--#include⟨*Filename*⟩

where ⟨*Filename*⟩ is a string enclosed by double quotations.

The line above is not an expression or a command of `Agda` itself, but is just a frontend. Thus there are some remarks.

- Although this directive starts with "`--`", it works with no problem (this line is not regarded as a comment).
- The suffixes "`.agda`" or "`.alfa`" can be omitted.
- Files are searched in
  - the current directory
  - the include path, set in the Emacs variable '`agda-include-path`'.
  - the file itself, if the absolute path is specified.
- An error occurs if the `Agda` program has no definitions and no `open` directives (See Section 4.3.2) but '`--#include`' directives.
- When an included file is not typechecked correctly, `Agda` does not load the current buffer and does not tell that.

4.3.2 Open expression

Names defined in other packages are referred with a dot like as ⟨*Package*⟩`.`⟨*Id*⟩. The `open` statement allows an `Agda` program to refer it to ⟨*Id*⟩ simply.

Let us see examples by use of the package we made in the last section.

- Without `open` statement

```
--#include "IntroLib.agda"

foo::IntroLib.List' IntroLib.Nat
                = IntroLib.con IntroLib.Nat
                               IntroLib.zer
                               (IntroLib.nil IntroLib.Nat)
```

- With `open` statement

```
--#include "IntroLib.agda"


open IntroLib use List', con, nil, Nat, zer

foo::List Nat = con Nat
                    zer
                    (nil Nat)
```

The syntax of open expressions is

`open` ⟨*Package*⟩ `use` ⟨*Id1*⟩ `::` ⟨*Type1*⟩`,` $\cdots$ `,` ⟨*Id n*⟩ `::` ⟨*Type n*⟩

where each $\langle Id\, i\rangle$, is defined in the package $\langle Package\rangle$. Type notations may be omitted.

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

```
open aPackage use loid=pacid
```

the name `pacid` defined in the package `aPackage` is referred to `loid` in the current buffer.

4.4 Packages with arguments

Let us see the following program:

```
package ListX (X::Set) where
  data List' = Nil | Con (x::X) (xs::List')

  tail'::List'->List' =
    \(l::List')->
      case l of
      (Nil     )-> Nil
      (Con x xs)-> xs
```

It is a package in which the list type with element of X is defined. The type X is the argument for this package, namely it comes from the outside of this package.

Inside the package, the type X is considered as an assumption. It takes effects throughout this package, but any definition or any declaration in this package do not know detailed informations on X even whether it exists or not.

This situation is similar to axioms in a logic. An axiom takes effects throughout the logic, but one does not want to know detailed informations about it. We will see concrete examples in Section 6.

By `open` declarations we can specialize these definitions:

```
data Bool' = true | false

open ListX Bool' use tail
```

The function `tail` is now from `List'` to `List'` on `Bool'`.

5 Basic operations for programming

Our aim in this section is to become familiar with the basic operations for developing Agda programs.

5.1 Making a code interactively

A significant function of Agda is an ability for making codes interactively. It plays an important role in proving theorems (Section 6), as well as it is useful for programming.

In this section we try communicating with Agda interactively. Agda has many commands to assist making Agda programs. (The possible commands are listed in Appendix A.)

Summary

- Making a code step by step, by refining goals.
- Templates for `let`, `case`, and abstractions.

5.1.1 The function `subList`

The program we make here is displayed in Figure 6 named "`subList.agda`". Expressions and directives appearing in it have been already introduced in the earlier part of this tutorial. Thus we concentrate operations here.

In the program, three functions are defined: for a type X,

- `append` : this function returns the concatenated list of its two arguments.
- `addElem` : for a given element x in X and a list ls in `List` (`List` X), it appends x to each element of ls;
- `subList` : for a given list xs in `List` X, make the list of all sublists of xs.

5.1.2 Editing a code

We focus on defining `subList` and we leave constructing the definitions of `append` and `addElem`. Just only memorize its type

```
append (|X::Set)::List' X -> List' X -> List' X
addElem (|X::Set):: X -> (List' (List' X)) -> (List' (List' X))
```

and go forward.

Let us input the following by hand (or cut-and-paste) and go forward. We can see the following lines in a Emacs buffer.

```
|-- subList.agda
|
|--#include "IntroLib.agda"
|open Intro use List', nil, con
|
|-- Preparing
|append (|X::Set)::List' X -> List' X -> List' X =
```

```
-- subList.agda

--#include "IntroLib.agda"
open Intro use List', nil, con

-- Preparing
append (|X::Set)::List' X -> List' X -> List' X =
    \(xs::List' X) -> \(ys::List' X) ->
        case xs of
          (nil      )-> ys
          (con x xs')-> con x (append xs' ys)
----

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

subList (|X::Set)::(List' X)  -> (List' (List' X)) =
    \(ys::List' X) ->
        case ys of
          (nil     )->
            con nil nil
          (con x xs)->
            let zs ::  List' (List' X)
                    = subList xs
            in  append zs (addElem x zs)
```

Figure 6: Source code

```
|    ?
|
|----
|
|addElem (|X::Set):: X -> (List' (List' X)) -> (List' (List' X)) =
|    ?
|
```

The functions `append` and `addElem` is bound to '`?`' (a metavariable or a goal), which means that the expression is not yet filled in.

Refining a goal. Let us type two lines, appending to lines we have input, in which the metavariables '`?`' appear in the type and the body.

```
|----
|
|addElem (|X::Set):: X -> (List' (List' X)) -> (List' (List' X)) =
|    ?
|
|subList (|X::Set):: ?
|        = ?
|
```

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

Next invoke the **Chase-load** command. `Agda` reads the file, as well as package files, and typechecks the file. Check that we are in the Proof-state (See Section 2) and that the the symbols '`?`' appearing in the code change to goals `{}0`, `{}1` and `{}2` respectively. We will refine goals step by step as follows.

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

The goal `{}1` is the type declaration of `subList'`. Since its type is a function type, first we can refine the goal to ⟨*Type1*⟩ `->` ⟨*Type2*⟩ for some ⟨*Type1*⟩ and ⟨*Type2*⟩. Fill `{}1` with the string "`? -> ?`" and try invoking the **Refine** command[16][17] `C-c C-r` on the goal.

What occurs? The goal `{}1` 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. At last, the goal `{}1` in the early version of the code is completely refined as follows:

Remark. The ordering of goals may be different between in our executions and in this tutorial. The ordering of goals depends on the order of operations, whether the buffer is loaded many times, versions of package files, and other

[16] We can also use the **Give** command, invoked by `C-c C-g`. These two commands have similar functions.

[17] As "`->`" itself is not an expression, a goal is not refined by filling with only "`->`".

reasons. However the ordering is for nothing but identification of goals, thus we need not worry about the differences of the ordering.

```
{}0
    ↓  fill
{?  -> ?}0
    ↓  [C-c C-r] (Refine)
{}2 -> {}3
    ↓  fill
{List'}2 -> {}3
    ↓  [C-c C-r]
List' {}4 -> {}3
    ↓  fill
List' {X}4 -> {}3
    ↓  [C-c C-r]
List' X -> {}3
    ↓  fill
List' X -> {List'}3
    ↓  [C-c C-r]
List' X -> List' {}5
    ↓  fill
List' X -> List' {List'}5
    ↓  [C-c C-r]
List' X -> List' (List' {}6)
    ↓  fill
List' X -> List' (List' {X}6)
    ↓  [C-c C-r]
List' X -> List' (List' X)
```

5.1.3 Templates, solving

Next we must fill the goal `{}2`, which is the function body. Fill the goal with the string "`ys`", which is the argument to `subList (X::Set)` and invoke **Abstraction** command by `C-c C-a` on the goal. Then the goal `{}2` changes to

```
= \(ys::List' X) ->
  {}8
```

This command gives a template of an abstraction expression and it fills automatically goals `Agda` can solve[18].

Like the abstraction command, **Case** command (`C-c C-c`) and **Let** command (`C-c C-l`) give templates respectively. Let us see the cocrete examples:

[18] If we want `Agda` to solve a goal explicitly, we should invoke **Solve** command with `C-c =` on the goal. With the latest version of `Agda` we do not have to invoke solve command in most cases, since `Agda` invokes that command for all goals automatically if refine, abstraction, load and some other commands are invoked.

```
subList (X::Set)::List' X -> List' (List' X)
        = \(ys::List' X) ->
          {}8
```

fill `{}8` with "ys" and `C-c C-c` (**Case**)

↓

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

fill `{}9` with "`con nil nil`" and `C-c C-r`

↓

```
subList (X::Set)::List' X -> List' (List' X)
        = \(ys::List' X) ->
          case ys of
          (nil     )->
            con nil nil
          (con x xs)->
            {}10
```

fill `{}10` with "`zs`" and `C-c C-l` (**Let**)

↓

```
subList (X::Set)::List' X -> List' (List' X)
        = \(ys::List' X) ->
          case ys of
          (nil     )->
            con nil nil
          (con x xs)->
            let zs::{}11
                  = {}12
            in  {}13
```

To fill the rest of goals is an easy exercise.

Remark. Even if in the Proof state, we can edit any part of programs except for deleting a goal. Especially, we often modify identifiers of variables given by `Agda` commands so that they express meanings of the variables. But `Agda` does not recognize the modifications automatically, so remember that we should invoke **Chase-load** command again after modifying a program.

Exercise

1. Complete the definition of `subLiet`.
2. We have left the definition of `append` and `addElem`. Check again its definition (Figure 6) and refine the goals.
3. Test the function `subList`. For some lists of some types apply the function to them and compute their values.

5.2 Other useful commands

Next we introduce some Agda commands to obtain states of the typechecker: commands to see constraints and contexts, to check a type of expression.

5.2.1 Go to an error

Typechecking a program stops if the program has some errors. By invoking **Go to error** with `C-c ‘` the cursor jumps to the location the first error occurs.

5.2.2 Show an unfolded type of a goal

In constructing a program/theorem, we often lose the type of a goal. Let us see the following example:

```
|--#include "IntroLib.agda"
|open IntroLib use List’, Nat
|
|LLNat::Set = List’ (List’ Nat)
|
|subThree::LLNat = ?
```

The type of the goal is off course `LLNat`. But what is `LLNat`? Place the cursor on the only goal in the above program and invoke the **Goal Type (Unfolded)** command by `C-c C-x C-r`. Then in the sub window, the answer

```
| ?0 :: List’ (List’ Nat)
```

is shown.

The command **Goal Type (Unfolded)** is very useful in dealing with Dependent Type Theory. Check the following program.

```
|--#include "IntroLib.agda"
|
|open IntroLib use Nat, Vec, three
|
|f1 (n::Nat)::Nat =
|    case n of
|        (zer   )-> three
|        (suc n’)-> n + three
|aVec (m::Nat)::Vec Nat (f1 m) =
|    case m of
|        (zer   )-> ?      -- Branch 1
|        (suc m’)-> ?      -- Branch 2
```

The command **Goal Type (Unfolded)** show the type of a goal which is unfolded as possible as Agda can at the context of the goal. When it is invoked at the first goal (the line indicated by “`Branch 1`”), the result is

```
| ?0 :: Vec Nat (suc@Nat (suc@Nat (sucNat zer@Nat)))
```

and when invoked at the second goal, the result is

```
| ?1 :: Vec Nat (suc@Nat (suc@Nat (suc@Nat (suc@_ m’))))
```

Thus that command helps us very much to construct objects of those types.

5.2.3 Show contexts

In making an `Agda` code we often forget some identifiers we have defined or which are read from packages. To check defined identifiers, we can use the command **Show contexts**.

Put the cursor on a goal after loading the following code.

```
|--#include "IntroLib.agda"
|
|open IntroLib use nat, suc, zer
|
|One = suc zer
|
|foo:: Nat = ?
|
|Two = suc One
```

Now invoke **Show contexts** Command by `C-c |` (vertical bar). Defined/opend identifiers and their types are displayed. They are built-in types, operators, functions and user defined types and functions.

```
|-EEE:---F1  Proof: first.agda       (Agda:run Ind)--L1--All-----
|(:) :: (A::Set) |-> (x::A) -> (xs::List A) -> List A
|Bool :: Set
|Char :: Set
...

|succ :: (x::Nat) -> Nat
|zero ::Nat
|-EEE:**-F1  Emacs: * Context *       (Agda:run Ind)--L1--All----
```

But notice the identifier `Two` is not presented. It is because the definition of `Two` follows the goal the cursor is put. (See also Section 3.2.5) Indeed, **Show contexts** command displays the context on which that command is invoked.

6 Theorem-proving in Agda

In this section, we demonstrate theorem-proving in basic logics and in First-order arithmetic using `Agda`. It is assumed the reader is familiar with Predicate logic.

We set up the following two objectives in this section.

- To understand a shallow embedding and to do it by using packages of `Agda`.

- To be able to prove theorems in the embedded basic logic using package implementation.

6.1 Introduction to dependent type theory

The `Agda` is an implementation of dependent type theory due to Per Martin-Löf. Type theory is used as a logical framework: Different theories can be represented in type theory. In this section, we represent several logical systems via interpretation due to Heyting. This is called a *shallow embedding*. We shall formalize proofs in dependent type theory, and then express them in `Agda`.

6.1.1 From informal proofs to formal proofs in type theory

As the first example, we shall treat implicational propositional logic. Let us transform informal proofs in the logic to formal proofs in type theory.

The idea of propositions as sets is to identify a proposition with the set of its proofs. This is Heyting's interpretation[2].

For implication, a proof of $A \supset B$ is a function (method, program) which takes each proof of A to a proof of B. Therefore, $A \supset B$ can be expressed by the function type $A \to B$.

Let Γ be a context consisting of formulas. $\Gamma \vdash A$ means that A is provable from the hypotheses Γ. We shall adapt natural deduction as a logical system. For each logical connective, there are two kinds of logical rules: Introduction rule(s) and Elimination rule(s). An Introduction rule corresponds to a direct proof, while an Elimination rule corresponds to an indirect proof.

Two rules for implication are as follows:

Introduction rule. $\Gamma, A \vdash B$ implies $\Gamma \vdash A \supset B$.

Elimination rule. $\Gamma \vdash A$ and $\Gamma \vdash A \supset B$ imply $\Gamma \vdash B$.

Both rules for implication are represented in type theory as follows:

Introduction rule.

$$\frac{\Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x.b : A \to B} \ (\to \text{Intro.})$$

The rule says that $A \supset B$ has a direct proof represented by $\lambda x.b : A \to B$, if for any given proof x of A, we can obtain a proof b of B.

Elimination rule.

$$\frac{\Gamma \vdash f : A \to B \quad \Gamma \vdash a : A}{\Gamma \vdash fa : B} \ (\to \text{Elim.})$$

The rule says that the B has an indirect proof represented by fa, if we have proofs represented as $a : A$ of A and $f : A \to B$ of $A \supset B$.

6.1.2 From formal proofs to agda expressions

Let us express the above formal proofs in `Agda`. "Propositions as sets" interpretation due to Heyting is expressed as the following definition.

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

We shall write implication by `=>`:

```
(=>) (X,Y::Prop) :: Prop = X -> Y
```

The expression `A => B` is the function type `A -> B` in `Agda`.

6.1.3 The first example: Implication

Look at the following proposition:

$$A \supset (B \supset A).$$

This proposition has a formal proof, $\lambda xy.x : A \to (B \to A)$, which has the following derivation in type theory.

$$\frac{\dfrac{x : A,\ y : B \vdash x : A}{x : A \vdash \lambda y.x \ : B \to A} \ (\to \text{Intro.})}{\vdash \lambda xy.x \ : A \to (B \to A)} \ (\to \text{Intro.})$$

The same proposition is expressed as `A => (B => A)` in `Agda`. We introduce the identifier `prop1` of the type `A => (B => A)`: The proof object below corresponds to the `Agda` expression of the above formal proof.

```
prop1 (A,B::Prop):: A => (B => A)
  = \(x::A)-> \(y::B)-> x
```

6.1.4 Predicates

Suppose that A is a set and P is a unary predicate on A. If $a : A$, then $B(a)$ is interpreted as the set of proofs of $B(a)$. Hence, P is interpreted as a unary function from A to the type of propositions. A function from some set to the type of propositions is called a *propositional function.*

Due to the interpretation, we define the type `Pred` in `Agda` as follows:

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

6.1.5 Universal quantification

Suppose that A is a set and P is a unary predicate on A. A direct proof of $\forall x : A.\, B(x)$ is a method which takes an object $x : A$ to a proof of $B(x)$. Thus $\forall x : A.\, B(x)$ is interpreted by the $\forall x : A.\, B(x)$ dependent funtion type.

Introduction rule. $\Gamma \vdash B(x)$ implies $\Gamma \vdash \forall x B(x)$, where x does not occur in Γ freely.

Elimination rule. $\Gamma \vdash \forall x B(x)$ and $a \in A$ imply $\Gamma \vdash B(a)$.

Both rules for universal quantification are represented in type theory as follows:

Introduction rule.

$$\frac{\Gamma,\ x : A \vdash p : B(x)}{\Gamma\ \vdash\ \lambda x.p\ : (\forall x : A.\ B(x))}\ (\forall\text{Intro.})$$

The rule says that $\forall x : A.\ B(x)$ has a direct proof represented by $\lambda x.p : (\forall x : A.\ B(x))$, if from any given proof x of A, we can obtain a proof p of $B(x)$.

Elimination rule.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash f : (\forall x : A.\, B(x))}{\Gamma \vdash f\, a : B(a)}\ (\forall\text{Elim.})$$

The rule says that the $B(a)$ has an indirect proof represented by $f\, a$, if we can have element a of A and a proof f of $\forall x : A.\, B(x)$.

The logical constant `Forall` is defined as follows:

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

We write `Forall A B` for $\forall x : A.\, B(x)$.

6.1.6 The second example

We shall prove the proposition:

$$\forall x\, (B \supset C) \supset (\forall x\, B \supset \forall x\, C),$$

where B, C are predicates. Let the variable x be of type D, i.e.,$x : D$.

This proposition has a formal proof,

$$\lambda fgx.(fx)(gx) : \forall x.\, (B \to C) \to (\forall x.\, B \to \forall x.\, C),$$

which has the following derivation in type theory, where Γ is $f : \forall x.\, (B \to C)$, $g : \forall x.\, B$, $x : D$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash f : \forall x.\,(B \to C) \quad \Gamma \vdash x : D}{\Gamma \vdash f\,x : B \to C}\,(\forall\text{Elim.}) \quad \dfrac{\Gamma \vdash g : \forall x.\,B \quad \Gamma \vdash x : D}{\Gamma \vdash g\,x : B}\,(\forall\text{Elim.})}{\Gamma \vdash (f\,x)(g\,x) : C}\,(\to\text{Elim.})}{f : \forall x.\,(B \to C), g : \forall x.\,B \vdash \lambda x.(f\,x)(g\,x) : \forall x.\,C}\,(\forall\text{Intro.})}{f : \forall x.\,(B \to C) \vdash \lambda gx.(f\,x)(g\,x) : \forall x.\,B \to \forall x.\,C}\,(\to\text{Intro.})}{\vdash \lambda fgx.(f\,x)(g\,x) : \forall x.\,(B \to C) \to (\forall x.\,B \to \forall x.\,C)}\,(\to\text{Intro.})$$

The above proof is expressed in `Agda` as follows.

```
prop2 (D::Set)(B,C ::Pred D)
  :: (Forall D (\(x::D)-> ((B x)=> (C x))))
     => ((Forall D B) => (Forall D C))
  =  \(f::Forall D ( \(x::D)-> B x => (C x)))->
      \(g::Forall D B)-> \(x::D)-> f x (g x)
```

6.2 Other logical connectives

6.2.1 Conjunction

Suppose that A and B are propositions. Let $A \wedge B$ be the conjunction of A and B.

Introduction rule. $\Gamma \vdash A$ and $\Gamma \vdash B$ imply $\Gamma \vdash A \wedge B$.

Elimination rule 1. $\Gamma \vdash A \wedge B$ implies $\Gamma \vdash A$.

Elimination rule 2. $\Gamma \vdash A \wedge B$ implies $\Gamma \vdash B$.

A direct proof of $A \wedge B$ is to take a pair consisting of the proofs of A and of B. This means that $A \wedge B$ is intrepreted as the cartesian product of A and B.

Introduction and Elimination rules for conjunction are expressed in type theory as follows:

Introduction rule.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B}{\Gamma \vdash < a, b >: A \wedge B}\,(\wedge\text{Intro.})$$

$< a, b >$ of type $A \wedge B$ expresses the pair of a and b.

Elimination rule 1.

$$\frac{\Gamma \vdash p : A \wedge B}{\Gamma \vdash e_1(p) : A}\,(\wedge\text{Elim. } 1)$$

e_1 is the projection returning the first element of p.

Elimination rule 2.

$$\frac{\Gamma \vdash p : A \wedge B}{\Gamma \vdash e_2(p) : B} \ (\wedge\text{Elim. } 2)$$

e_2 is the projection returning the second element of p.

We shall define the logical constant `&&` in `Agda` to express conjunction as follows:

```
(&&) (X,Y::Prop) :: Prop
   = sig{fst :: X; snd :: Y;}
```

The `fst` and `snd` are the first and second parameters. We shall write $A \wedge B$ as `A && B`.

In `Agda`, a pair is expressed as follows.

```
pair (X,Y::Prop)(x::X)(y::Y) :: X && Y
    = struct {fst = x; snd = y;}
```

In `Agda`, projections are expressed as follows.

```
pair_fst (X,Y::Prop)(xy:: X && Y) :: X
    = xy.fst
pair_snd (X,Y::Prop)(xy:: X && Y) :: Y
    = xy.snd
```

6.2.2 Disjunction

Suppose that A and B are propositions. Let $A \vee B$ be the disjunction of A and B.

Introduction rule 1. $\Gamma \vdash A$ implies $\Gamma \vdash A \vee B$.

Introduction rule 2. $\Gamma \vdash B$ implies $\Gamma \vdash A \vee B$.

Elimination rule. $\Gamma \vdash A \vee B$ and $\Gamma, A \vdash C$ and $\Gamma, B \vdash C$ imply $\Gamma \vdash C$.

A disjunction is constructively true if and only if we can prove one of the disujuncts. So a proof of $A \vee B$ is either a proof of A or a proof of B. Thus $A \vee B$ is interpreted as the *disjoint union*.

Introduction and Elimination rules for disjunction are expressed in type theory as follows:

Introduction rule 1.

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash inl(a) : A \vee B} \ (\vee\text{Intro. } 1)$$

inl is the embedding of A into $A \vee B$. The name *inl* stands for "in the left component".

Introduction rule 2.

$$\frac{\Gamma \vdash b : B}{\Gamma \vdash inr(b) : A \vee B} \ (\vee\text{Intro. } 2)$$

inr is the embedding of B into $A \vee B$. The name *inr* stands for "in the right component".

Elimination rule.

$$\frac{\Gamma \vdash p : A \vee B \quad \Gamma,\, x : A \vdash e : C(inl(x)) \quad \Gamma,\, y : B \vdash f : C(inr(y))}{\Gamma \vdash when(e, f, p) : C(p)}$$

where $A : Prop$, $B : Prop$, and C is a propositional function from $A \vee B$ to $Prop$.

The function *when* is defined for $a : A$ and $b : B$ by the equations

$$when(e, f, inl(a)) = e(a), \quad when(e, f, inr(b)) = f(b).$$

If $p : A \vee B$, $e : C(inl(x))$ and $f : C(inr(y))$ represent proofs of sequents $\Gamma \vdash A \vee B$, $\Gamma, A \vdash C$ and $\Gamma, B \vdash C$, respectively, then $when(e, f, p)$ is a proof of $\Gamma \vdash C$ via $\vee$-elimination.

We shall define the logical constant '`||`' in `Agda` to express disjunction as follows:

```
data (||) (X,Y::Prop)
   = data inl (x::X) | inr (y::Y)
```

We shall write disjunction $A \vee B$ as `A || B`. In `Agda`, the *when* function is expressed as follows.

```
when (A, B::Prop)(C::(A || B) -> Prop)
     (e::(x::A) -> C (inl@_ x))
     (f::(y::B) -> C (inr@_ y))
     (p::A || B)
   :: C p
   = case p of {
       (inl x) -> e x;
       (inr y) -> f y;}
```

The *when'* function below is a special case of *when*, which corresponds to $\vee$-elimination for propositional logic.

```
when' (A,B,C::Prop)(e::A => C)(f::B => C)
    :: (A || B) => C
    = \(p::A || B) ->
      case p of {
        (inl x) -> e x;
        (inr y) -> f y;}
```

6.2.3 Falsity

Falsity is a proposition with no proof. Therefore it is interpreted as the empty set. Elimination rule for falsity is as follows:

Elimination rule. $\Gamma \vdash \perp$ imply $\Gamma \vdash C$.

for an arbitrary object $C : Prop$. Hence, we define `Absurd` to be the empty set in `Agda`:

```
data Absurd =
```

In a proof by contradiction, we often construct a proof of falsity under a contradictory assumptions (c.f. Subsection 6.2.5).

Let `h::Absurd`. Then the expression

```
case h of {}
```

is an object of any type. The case expression '`case h of {}`' represents a proof of any proposition derived from $\perp$. This corresponds to the Elimination rule for falsity in type theory.

$$\frac{\Gamma \vdash h : \perp}{\Gamma \vdash case\ h\ of\ \{\} : C}\ (\perp\text{Elim.})$$

6.2.4 Truth

Truth is a provable proposition. Any non-empty set can represent Truth, but we define `Taut` to be a singleton set in `Agda`:

```
data Taut = tt
```

The unique object of type `Taut` is `tt@_`. This corresponds to the unique element in the singleton set.

6.2.5 Negation

Suppose A be a set. In intuitionistic logic, it is common to define $\neg A$ as $A \supset \perp$. Hence we shall adapt this definition for `Not` in `Agda`:

```
Not (X::Prop) :: Prop = X => Absurd
```

Let a and f be proofs of A and of $\neg A$, respectively. Then $f\,a$ is the proof of falsity. As we discussed in Subsection 6.2.3, the expression

```
case f a of {}
```

can be an object of any type. Thus we use '`case f a of {}`' to denote the proof of any proposition derived from the contradiction A and $\neg A$.

6.2.6 Existential quantification

Suppose that A is a set and B is a unary predicate on A. The proposition $\exists x : A. B(x)$ is constructively true if and only if we can find an object $a : A$ and a proof of $B(a)$ Thus $\exists x : A. B(x)$ is interpreted as the *dependent co-product*, or *dependent sum* denoted as $\Sigma_{x:A} B(x)$.

Introduction rule. $\Gamma \vdash B(a)$ implies $\Gamma \vdash \exists x B(x)$.

Elimination rule. $\Gamma \vdash \exists x B(x)$ and $\Gamma,\ B(x) \vdash C$ imply $\Gamma,\ \exists x B(x) \vdash C$, where x does not occur in Γ freely.

Introduction and Elimination rules for existential quantification are expressed in type theory as follows:

Introduction rule.

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash p : B(a)}{\Gamma \vdash < a, p >: (\exists x : A. B(x))} \ (\exists\text{Intro.})$$

Elimination rule.

$$\frac{\Gamma,\ x : A,\ y : B(x) \vdash d(x, y) : C(< x, y >) \quad \Gamma \vdash q : (\exists x : A. B(x))}{\Gamma \vdash \mathit{split}(d, q) : C(q)} \ (\exists\text{Elim.})$$

where $A : \mathit{Set}$, B is a unary predicate on A, and C is a propositional function from $\Sigma_{x:A} B(x)$ to *Prop*.

The function *split* is defined for $a : A$ and $p : B(a)$ by the equations

$$\mathit{split}(d, < a, p >) = d(a, p)$$

Let $a : A$. If $q : (\exists x : A. B(x))$ and $d(x, y) : C(< x, y >)$ represent proofs of sequents $\Gamma \vdash \exists x B(x)$ and $\Gamma,\ B(x) \vdash C$, respectively, then $\mathit{split}(d, < a, p >)$ is a proof of C via $\exists$-elimination.

We shall define the logical constant `Exist` in `Agda` as follows:

```
Exist (A::Set)(B::Pred A) :: Prop
   = sig{fst :: A; snd :: B fst;}
```

Then we write $\exists x : A. B(x)$ as `Exist A B`.

In `Agda`, a dependent pair is expressed as follows.

```
dep_pair (X::Set)(Y::Pred X)(x::X)(y::Y x) :: Exist X Y
   = struct {fst = x; snd = y;}
```

In `Agda`, projections are expressed as follows.

```
dep_fst (X::Set)(Y::Pred X)(xy::Exist X Y) :: X
   = xy.fst
dep_snd (X::Set)(Y::Pred X)(xy::Exist X Y)
   :: Y (dep_fst X Y xy)
   = xy.snd
```

In Agda, the *split* function is expressed as follows.

```
split (X::Set)(Y::Pred X)
     (Z::Exist X Y -> Prop)
     (d::(x::X) -> (y::Y x) -> Z (dep_pair X Y x y))
     (p::Exist X Y)
  :: Z p
  = d p.fst p.snd
```

The *split'* function below is a special case of *split*, which corresponds to $\exists$-elimination for a proposition independent of the proof.

```
split' (X::Set)(Y::Pred X)(Z::Prop)
    :: ((x::X) -> Y x -> Z) -> Exist X Y -> Z
    = \(f::(x::X) -> (x'::Y x) -> Z) ->
      \(p::Exist X Y) -> f p.fst p.snd
```

6.2.7 Logical equivalence

We shall express the logical equivalence $A \equiv B$ as `A <=> B`. The equivalence $A \equiv B$ is defined as $(A \supset B) \wedge (B \supset A)$. Hence we can make the following definition:

```
(<=>) (X,Y::Prop) :: Prop = (X => Y) && (Y => X)
```

The proposition $A \equiv B$ is expressed as `A <=> B`.

6.3 Equality

6.3.1 Relations

Let A be a set. The equality $=_A$ is a binary relation on A. The type of such binary relations is expressed by `Rel` as follows:

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

6.3.2 Equivalence relation

Equivalence relation satisfies three axioms; the reflexivity, the symmetry and the transitivity. They are expressed in Agda as follows:

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

6.3.3 Equality

Equality is an equivalence relation satisfying the axiom of substitutivity. It is expressed in Agda as follows:

```
Substitutive (X::Set)(R::Rel X) :: Type
  = (P::Pred X) -> (x1::X) -> (x2::X) ->
    R x1 x2 -> P x1 -> P x2
```

Let us call the laws of equality three axioms of equivalence relation plus the axiom of substitutivity.

6.3.4 Identity

The basic equality relation is the identity[19]. We define the identity `Id` in `Agda` by `idata` type construction as follows.

```
Id (X::Set) :: Rel X = idata ref (x::X) :: _ x x
```

where `ref` is a constructor. The name stands for the reflexivity.

Let `A::Set` and `a,b::A`. Then the type `Id A a b` represents the proposition $a =_A b$. The above idata type declaration says that its object (proof) must have the form of `ref@_ c` for some `c::A`, and further its type is `Id A c c`. So `Id A a b` is non-empty if and only if `a` and `b` are convertible under `Agda`'s conversion rule[20]. We can prove the laws of equality `refId`, `symId`, `tranId`, `substId` for `Id` as follows in `Agda`:

```
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

tranId(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
substId (X::Set) :: Substitutive X (Id X)
  =  \(P::Pred X)-> \(x1::X)-> \(x2::X)->
     \(h::Id X x1 x2)->
     \(h'::P x1)-> case h of (ref x)-> h'
```

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 function `mapId` below formulates the extensionality with the identity as the equality.

```
mapId(X :: Set)(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)
```

Finally, we shall define the package `LogicLib.agda` containing the definitions of logical constants and the definitions of the identity and related functions introduced in this subsection.

```
-- LogicLib.agda
```

[19] `Id` is in `Identity proof sets` in the SET package(See Section 4).
[20] β-conversion and, for functions and records, η-conversion.

```
package LogicLib where

  -- Section 6.1.2
  Prop :: Type = Set

  (=>)(X,Y::Prop) :: Prop = X -> Y

  -- Section 6.1.4
  Pred (X::Set) :: Type = X -> Prop

  -- Section 6.1.5
  Forall(D::Set)(P::Pred D) :: Prop = (x::D) -> P x

  -- Section 6.2.1
  (&&)(X,Y::Prop) :: Prop
    = sig{fst :: X; snd :: Y;}

  pair (X,Y::Prop) (x::X) (y::Y):: X && Y
    = struct
        fst:: X = x
        snd:: Y = y

  pair_fst (X,Y::Prop)(xy:: X && Y) :: X
    = xy.fst

  pair_snd (X,Y::Prop)(xy:: X && Y) :: Y
    = xy.snd

  -- Section 6.2.2
  data (||)(X,Y::Prop)
    = inl (x::X) | inr (y::Y)

  when (X,Y::Prop)(C:: (X || Y) -> Prop)
        (e:: (x::X) -> C (inl x))
        (f:: (y::Y) -> C (inr y))
        (p:: X || Y)
        :: C p
    = case p of
      (inl x)-> e x
      (inr y)-> f y

  when' (X,Y,Z::Prop)(e::X => Z)(f::Y => Z)
    :: (X || Y) => Z
    = \(p::X || Y)->
        case p of
        (inl x)-> e x
        (inr y)-> f y

  -- Section  6.2.3
  data Absurd =

  -- Section 6.2.4
  data Taut = tt
```

```
-- Section 6.2.5
Not (X::Prop) :: Prop = X => Absurd

-- Section 6.2.6
Exist (X::Set)(Y::Pred X) :: Prop
  = sig{fst :: X; snd :: Y fst;}

dep_pair (X::Set) (Y::Pred X) (x::X) (y::Y x) :: Exist X Y
  = struct
      fst:: X = x
      snd:: Y fst = y

dep_fst (X::Set) (Y::Pred X) (xy::Exist X Y) :: X
  = xy.fst

dep_snd (X::Set) (Y::Pred X) (xy::Exist X Y) :: Y (dep_fst X Y xy)
  = xy.snd

split (X::Set) (Y::Pred X)
      (Z::Exist X Y -> Prop)
      (d:: (x::X) -> (y:: Y x) -> Z (dep_pair X Y x y))
      (p::Exist X Y)
      :: Z p
  = d p.fst p.snd

split' (X::Set) (Y::Pred X) (Z::Prop)
      ::((x::X) -> Y x -> Z) -> Exist X Y -> Z
  = \(f::(x::X) -> (x'::Y x) -> Z) ->
      \(p::Exist X Y) -> f p.fst p.snd

-- Section 6.2.7
(<=>)(X,Y::Prop) :: Prop = (X => Y) && (Y => X)

-- Section 6.3.1
Rel (X::Set) :: Type = X -> X -> Prop

-- Section 6.3.2
Reflexive (X::Set)(R::Rel X) :: Prop = (x::X) -> R x x

Symmetrical (X::Set)(R::Rel X) :: Prop
  = (x1::X) -> (x2::X) -> R x1 x2 -> R x2 x1

Transitive (X::Set)(R::Rel X) :: Prop
  = (x1::X) -> (x2::X) -> (x3::X) ->
    R x1 x2 -> R x2 x3 -> R x1 x3

-- Section 6.3.3
Substitutive (X::Set)(R::Rel X) :: Type
  = (P::Pred X) -> (x1::X) -> (x2::X) ->
    R x1 x2 -> P x1 -> P x2

-- Section 6.3.4
Id (X::Set) :: Rel X
  = idata ref (x::X) :: _ x x
```

```
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

tranId(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

substId (X::Set)::Substitutive X (Id X)
  = \(P::Pred X)-> \(x1::X)-> \(x2::X)->
    \(h::Id X x1 x2)->
    \(h'::P x1)-> case h of (ref x)-> h'

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)
```

Thus we can use the above definitions from other files by simply opening the `LogicLib` package. For an example, we can write the two examples are written in a separate `Agda` file as follows.

```
--#include "LogicLib.agda"

open LogicLib use Prop, Pred, Forall, (&&), (=>), (<=>)

prop1 (A,B::Prop) :: A => (B => A)
    = \(h::A)-> \(h'::B)-> h

prop2 (X::Prop)(A,B::Pred X)
    :: (Forall X (\(x::X)-> ((A x)=> (B x))))
    => ((Forall X A) => (Forall X B))
    =  \(h::Forall X ( \(x::X)-> A x => (B x)))->
       \(h'::Forall X A)-> \(x::X)-> h x (h' x)
```

6.4 Examples

We shall explain proofs of the following kinds:

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

6.4.1 Intuitionistic Propositional logic

We shall prove the following proposition:

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

In `Agda`, we express the proposition as follows:

```
((A && B) => C) <=> (A => (B => C)).
```

The complete proof of the proposition in `Agda` is in Figure 7.

```
--#include "LogicLib.agda"

open LogicLib use Prop, (&&), (=>), (<=>)

prop3 (A,B,C::Prop)::((A && B) => C) <=> (A => (B => C))
  = struct
      fst:: (A && B) => C -> A => (B => C)
        = \(h::(A && B) => C)->
                \(h'::A)-> \(h0::B)-> h (struct
                                         fst:: A = h'
                                         snd:: B = h0
                                        )
      snd:: A => (B => C)-> (A && B) => C
        = \(h::A -> B -> C)->
            \(h'::A && B)-> h h'.fst h'.snd
```

Figure 7: Source code

Let us construct the proof step by step: Input the earlier part of the program and make the goal `0` by **Chase-load** as follows:

```
|--#include "LogicLib.agda"
|
|open LogicLib use Prop, (&&), (=>), (<=>)
|
|prop3 (A,B,C::Prop) :: ((A && B) => C) <=> (A => (B => C))
|  = {}0
|
```

Proving the proposition

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

is to prove two propositions

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

and

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

The **Intro** command invoked by `C-c C-i` provides the prototype of the object to fill in. Invoke **Intro** command at the goal 0 and obtain the following:

```
|  prop3 (A,B,C::Prop) :: ((A && B) => C) <=> (A => (B => C))
|    = struct
|      fst ::(A && B) => C -> A => (B => C)
|        = { }0
|      snd ::A => (B => C) -> (A && B) => C
|        = { }1
|
```

The subgoals 0, 1 have the following types (Note that the goals are displayed in the reverse order in Sub window.).

```
|?1 :: A => (B => C) ->  (A && B) => C
|?0 :: (A && B) => C ->  A => (B => C)
```

We have obtained the new goals 0 and 1. We shall start with the new goal 0.

The corresponding proposition has a formal proof written as follows. First check the lower part of the proof:

$$\dfrac{\dfrac{\dfrac{h : (A \wedge B) \to C,\ h' : A,\ h_0 : B \vdash h < h', h_0 >: C}{h : (A \wedge B) \to C,\ h' : A \vdash \lambda h_0.h < h', h_0 >: B \to C}(\to \text{Intro.})(3)}{h : (A \wedge B) \to C \vdash \lambda h' h_0.h < h', h_0 >: A \to (B \to C)}(\to \text{Intro.})(2)}{\vdash \lambda h h' h_0.h < h', h_0 >: ((A \wedge B) \to C) \to (A \to (B \to C))}(\to \text{Intro.})(1)$$

Let Γ be $h : (A \wedge B) \to C$, $h' : A$, $h_0 : B$. Then we have the rest of the proof as follows.

$$\dfrac{\Gamma \vdash h : (A \wedge B) \to C \quad \dfrac{\Gamma \vdash h' : A \quad \Gamma \vdash h_0 : B}{\Gamma \vdash < h', h_0 >: A \wedge B}(\wedge\text{Intro.})(5)}{\Gamma \vdash h < h', h_0 >: C}(\to \text{Elim.})(4)$$

We construct this proof in `Agda` from the bottom of the derivation towards the top. Invoke **Intro** command `C-c C-i` at the goal 0. Then we obtain the following:

```
|  prop3 (A,B,C::Prop) :: ((A && B) => C) <=> (A => (B => C))
|    = struct
|      fst ::(A && B) => C -> A => (B => C)
|        =  \(h:: (A && B) => C)->
|           {}2
|      snd  ::A => (B => C)-> (A && B) => C
|        =  {}1
|
```

where the subgoal 2 has the following type.

```
?2 :: A => (B => C)
```

This corresponds to $(\to$ Intro.)(3) in the formal proof.

Invoke **Intro** command `C-c C-i` at the goal 2, and we obtain this:

```
|  prop3 (A,B,C::Prop) :: ((A && B) => C) <=> (A => (B => C))
|    = struct
|      fst ::(A && B) => C -> A => (B => C)
|        =  \(h:: (A && B) => C)->
|            \(h'::A)->
|            {}3
|      snd ::A => (B => C)-> (A && B) => C
|        = {}1
|
```

where the subgoal 3 has the following type.

```
?3 :: B => C
```

This corresponds to ($\rightarrow$ Intro.)(2) in the formal proof.

Invoke **Intro** command `C-c C-i` at the goal 3, and we obtain this:

```
|  prop3 (A,B,C::Prop) :: ((A && B) => C) <=> (A => (B => C))
|    = struct
|      fst ::(A && B) => C -> A => (B => C)
|        =  \(h:: (A && B) => C)->
|            \(h'::A)->
|             \(h0::B)->
|             {}4
|      snd ::A => (B => C)-> (A && B) => C
|        = {}1
|
```

where the subgoal 4 has the following type.

```
?4 :: C
```

This corresponds to ($\rightarrow$ Intro.)(1) in the formal proof.

Now let us work on the goal 4: We shall construct an object of type C using the objects `h`, `h'` and `h0`.

If some object `e :: A && B` is available, then application `h e` is of type `C`. Thus input `h` in the goal 4 and invoke **Refine** command `C-c,C-r`. Now `Agda` checks that `h` can produce an object of type `C`, provided that such an `e` can be given; and it shows the new goal 5:

```
|  prop3 (A,B,C::Prop) :: ((A && B) => C) <=> (A => (B => C))
|    = struct
|      fst ::(A && B) => C -> A => (B => C)
|        =  \(h::(A && B) => C)->
|            \(h'::A)->
|             \(h0::B)->
|             h {}5
|      snd ::A => (B => C)-> (A && B) => C
|        = {}1
|
```

where the subgoal 5 has the following type.

```
?5 :: A && B
```

This corresponds to ($\to$ Elim.)(4) in the formal proof.

Let us construct a concrete object of type `A && B`. Invoke **Intro** command $\boxed{\texttt{C-c C-i}}$ at goal 5: Then it provides the prototype of the object of this type, and shows the following:

```
| prop3 (A,B,C::Prop) :: ((A && B) => C) <=> (A => (B => C))
|   = struct
|     fst ::(A && B) => C -> A => (B => C)
|       =  \(h::(A && B) => C)->
|           \(h'::A)->
|            \(h0::B)->
|                 h (struct
|                      fst ::A
|                        = {}5
|                      snd :: B
|                        = {}6
|                    )
|     snd  ::A => (B => C)-> (A && B) => C
|       = {}1
|
```

where the subgoals 5 and 6 have the following type.

```
?6 :: B
?5 :: A
```

Input `h'` into the goal 5 and invoke **Refine** command $\boxed{\texttt{C-c C-r}}$. Similarily, input `h0` into the goal 6 and invoke the same **Refine** command:

```
| prop3 (A,B,C::Prop) :: ((A && B) => C) <=> (A => (B => C))
|   = struct
|     fst ::(A && B) => C -> A => (B => C)
|       =  \(h::(A && B) => C)->
|           \(h'::A)->
|            \(h0::B)->
|                 h (struct
|                      fst ::A
|                        = h'
|                      snd ::B
|                        = h0
|                   )
|     snd  ::A => (B => C)-> (A && B) => C
|       = { }1
|
```

This corresponds to ($\wedge$Intro.)(4) in the formal proof. It completes the definition of `fst`.

We shall delete unnecessary spaces to make it look better.

```
| prop3 (A,B,C::Prop) :: ((A && B) => C) <=> (A => (B => C))
|   = struct
|       fst ::(A && B) => C -> A => (B => C)
```

```
|          =  \(h::(A && B) => C)-> \(h'::A)-> \(h0::B)->
|              h (struct {fst ::A = h'; snd ::B = h0;})
|        snd  ::A => (B => C)-> (A && B) => C
|    = {}1
|
```

Let us turn to the goal 1: Invoke **Intro** command `C-c C-i` at the goal 1 twice. Then we obtain the following:

```
|  prop3 (A,B,C::Prop) :: ((A && B) => C) <=> (A => (B => C))
|    = struct
|        fst ::(A && B) => C -> A => (B => C)
|          =  \(h::(A && B) => C)-> \(h'::A)-> \(h0::B)->
|             h (struct {fst ::A = h'; snd ::B = h0;})
|        snd ::A => (B => C)-> (A && B) => C
|          =  \(h:: A -> B -> C)->
|              \(h'::A && B)->
|              { }9
|
```

where the subgoal is of the following type.

```
?9 :: C
```

Input `h` in the goal 9 and invoke **Refine** command `C-c,C-r`. Now `Agda` checks the type of `h` and shows new goals 9 and 10:

```
snd ::A => (B => C)-> (A && B) => C
  =  \(h:: A -> B -> C)->
      \(h'::A && B)-> h {}9
                        {}10
```

where the subgoal is of the following type.

```
?10 :: B
?9 :: A
```

Due to the definition of `A && B`, `h'.fst::A` and `h'.snd::B` are the objects for goals 9 and 10, respectively.

Input `h'.fst` into the goal 9 and invoke **Refine** command `C-c C-r`. Similarily, input `h'.snd` into the goal 10 and invoke the same **Refine** command:

```
snd ::A => (B => C)-> (A && B) => C
  =  \(h:: A -> B -> C)->
      \(h'::A && B)-> h h'.fst
                        h'.snd
```

It completes the definition of `snd`.

6.4.2 Exercises.

Prove the following using in `Agda`.

1. $A \supset (\neg(\neg A))$

```
Exer1 (A::Prop) :: A => (Not (Not A))
```

2. $((\neg A) \vee B) \supset (A \supset B)$

```
Exer2 (A,B::Prop)::((Not A) || B) =>  (A => B)
```

3. $((\neg A) \wedge (\neg B)) \supset \neg(A \vee B)$ (Hint: Use **Abstraction**.)

```
Exer3 (A,B::Prop)::(Not A) && (Not B) => Not (A || B)
```

4. $(A \supset B) \supset ((\neg B) \supset (\neg A))$

```
Exer4 (A,B::Prop)::(A => B) => ((Not B) => (Not A))
```

6.4.3 Classical propositional logic

In this subsection, we shall demonstrate a simple way to prove a tautology in the classical propositional logic using `Agda`.

The law of excluded middle is a formula of the form $P \vee \neg P$, where P is a some formula. In intuitionistic logic, the law of excluded middle is not generally provable.

In general, there are two ways to prove a classical tautology A in `Agda`: One is to use Glivenko's theorem in propositional logic: A is provable in classical logic if and only if $\neg\neg A$ is provable in Intuitionistic logic.

The other is to define a package and to use a `postulate` representing the classical principle such as the law of excluded middle. In this subsection, we shall take the latter approach. Now we shall prove the proposition

$$(A \supset B) \equiv ((\neg A) \vee B).$$

In `Agda`, we express the proposition as follows:

```
(A => B) <=> (Not A || B).
```

Since we like to use the law of excluded middle, let us open a new file and define the package named `Classical` as follows:

```
|--#include "LogicLib.agda"
|
|package Classical where
|
|  open LogicLib use Prop, (||), inr, inl, Not, (=>), (<=>)
|  postulate em :: (X::Prop) -> (X || Not X)
|
|  prop4 (A,B::Prop)::A => B <=> (Not A || B)
|    = {}0
|
|
```

The above `postulate` declares a new primitive constant `em` of the given type. Assume `P :: Prop`. Then `em P` represents a proof of the proposition `P || Not P`.

The complete proof of the proposition in `Agda` is in Figure 8. Let us construct

```
--#include "LogicLib.agda"

package Classical where

  open LogicLib use Prop, (||), inl, inr, Not, (=>), (<=>)
  postulate em :: (X::Prop) -> (X || Not X)

  prop4 (A,B::Prop)::A => B <=> (Not A || B)
    = struct
        fst:: A => B -> Not A || B
         = \(h::A -> B)->
                case em A of
                (inl a)-> inr (h a)
                (inr na)->inl na
        snd:: Not A || B -> A => B
         = \(h::Not A || B)->
                \(a::A)->
                    case h of
                    (inl na)-> case na a of { }
                    (inr b)->  b
```

Figure 8: Source code

the proof of prop4. Invoke **Intro** at the goal 0, and we obtain the following:

```
|  prop4 (A,B::Prop):: A => B <=> (Not A || B)
|     = struct
|         fst ::A => B -> (Not A || B)
|           = {}0;
|         snd ::(Not A || B)-> A => B
|           = {}1
|
```

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

```
|  prop4 (A,B::Prop):: A => B <=> (Not A || B)
|     = struct
|         fst ::A => B -> (Not A || B)
|           = \(h:: A -> B)->
|             {}2
|         snd ::(Not A || B)-> A => B
|           = {}1
|
```

where the subgoal 2 has the following type.

```
?2 :: Not A || B
```

We like to find an object of type `(Not A)` or `B`. Let's use the law of excluded middle of `em A`.

Since `em A` has the disjunctive type, we can analyse it by cases. Input `em A` and invoke **Case**, and we obtain the following:

```
fst ::A => B -> (Not A || B)
  = \(h:: A -> B)->
    case em A of
      (inl x)->
        {}3
      (inr y)->
        {}4
```

where the subgoals 3 and 4 have the following type.

```
?4 :: Not A || B
?3 :: Not A || B
```

Here `x` and `y` have types `A` and `Not A`, respectively. We replace them by new better names: Edit `x` and `y` to `a` and `na`, respectively, and invoke **chase-load** command: Then we obtain the following:

```
fst ::A => B -> (Not A || B)
  = \(h:: A -> B)->
    case em A of
      (inl a)->
        {}3
      (inr na)->
        {}4
```

Since `h` is a function of type `A -> B`, the application `h a` has type `B`. Then `inr (h a)` has type `Not A || B`. Input `inr (h a)` into the goal 3 and invoke **Refine** command.

```
fst ::A => B -> (Not A || B)
  = \(h:: A -> B)->
    case em A of
      (inl a)->
        inr (h a)
      (inr na)->
        {}4
```

Similarly, input `inl na` to the goal 4 and invoke **Refine** command.

```
|  prop4 (A,B::Prop):: A => B <=> (Not A || B)
|     = struct
|         fst ::A => B -> (Not A || B)
|           = \(h:: A -> B)->
|             case em A of
|               (inl a)->
```

```
|                 inr (h a)
|               (inr na)->
|                 inl na
|         snd ::(Not A || B)-> A => B
|           = {}1
|
```

It completes the proof of `fst`.

Let us work on the goal 1: Invoke **Intro** command twice, and we obtain the following:

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

where the subgoal 5 has the following type.

```
?5 :: B
```

Since `h` has disjunctive type `Not A || B`, we analyse by cases whether the `h` proves `Not A` or `B`. Input `h` into the goal 5 and invoke **Case** command:

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

where the subgoals 6 and 7 have the following types.

```
?7 :: B
?6 :: B
```

`x` and `y` are proofs of `Not A` and `B`, respectively. The goal 7 is filled with `y` itself. Input `y` to the goal 7 and invoke **Refine** command.

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

Now we shall work on the goal 6. Here we need absurdity elimination. Let `h'::A` and `x::Not A`. The expression `x h'` has type `Absurd`. We can check it as follows: Input `x h'` into the goal 6 and invoke **Infer type** command $\boxed{\texttt{C-c :}}$. Then it shows in the Sub Window the following:

```
|
|Absurd
|
```

Now invoke **Case** at the goal 6. and we obtain the following:

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

This completes the proof of `prop4`.

Remark. The law of excluded middle for decidable predicates. If one wants to assume the law of excluded middle for any proposition, then he/she needs the postulate representation. However, the law of excluded middle is provable for specific propositions, which are sometimes called decidable.

Let "==" be the equality between natural numbers defined in Section 3.4.4. Then this equality is decidable, that is `x == y || Not(x == y)` is provable in `Agda`, thus the following program is correct:

```
--#include "IntroLib.agda"
--#include "LogigLib.agda''

open IntroLib use Bool', true, false, Nat, zer, suc, (==)
open LogicLib use Taut, Absurd, tt, Prop

TrueBool (p::Bool') :: Prop
    = case p of
      (true) -> Taut
      (false) -> Absurd

not (x::Bool') :: Bool'
  = case x of
      (true) -> false
      (false) -> true

or (x::Bool')(y::Bool') :: Bool'
  = case x of
      (true) -> true
      (false) -> y

lem (x,y::Nat):: TrueBool (or (x == y) (not (x == y)))
  = let pem (p::Bool'):: True (or p (not p))
        pem  = case p of
               (true )-> tt
               (false)-> tt
    in  pem (x == y)
```

6.4.4 Exercises.

Prove the following using in Agda.

1. $(\neg(\neg A)) \supset A$

```
Exer5 (A::Prop) :: (Not (Not A)) => A
```

6.4.5 Intuitionistic first-order predicate logic

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

Let us open a new file and define the package named `Predicate` with parameters `D::Prop` and `d0::D` as follows:

```
|--#include"LogicLib.agda"
|
|package Predicate (D::Set) (d0::D) where
|
|  open LogicLib use Prop, Pred, Forall, Prop, (<=>), (=>)
```

Remember the definition of `Pred`:

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

Our embedding requires the domain of individuals necessary for semantics of first-order predicate logic. Thus we have introduced the parameter `D::Set` to denote the domain.

Let us prove the logical equivalence

$$\exists x P \equiv P$$

where P is a unary predicate without free occurrence of the variable x.

Expressing a proposition with a dummy quantifier. How do we express the proposition $\exists x P$ for P without free occurrence of variable x in Agda? Here is the solusion: Let `P::Prop`. Then `\(x::D)-> P` represents a constant function of the type `Pred D`. Now the $\exists x P$ is expressed as `Exist D (\(x::D)-> P)`.

Expressing a non-empty domain. The logical equivalence

$$\exists x P \equiv P$$

is true if and only if the domain is non-empty. To denote the non-emptiness of the domain, we have introduced the parameter `d0 :: D` to the package.

Thus the above proposition is expressed as follows:

```
|--#include"LogicLib.agda"
|
|package Predicate (D::Set) (d0::D) where
|
```

```
|  open LogicLib use Prop, Pred, Forall, Prop, (<=>), (=>)
|
|  pred5 (P::Prop) :: (Exist D (\(x::D) -> P)) <=> P
|    = {}0
|
```

The complete proof of the proposition in `Agda` is in Figure 9. Let us work on

```
--#include"LogicLib.agda"

package Predicate (D::Set) (d0::D) where

  open LogicLib use Prop, Pred, Forall, Prop, (<=>), (=>)

  pred5 (P::Prop) :: (Exist D (\(x::D) -> P)) <=> P
    = struct
        fst::Exist D (\(x::D) -> P) -> P
          = \(h::Exist D (\(x::D) -> P)) ->
                h.snd
        snd::P -> Exist D (\(x::D) -> P)
          = \(h::P) ->
            struct
                fst::D = d0
                snd::P = h
```

Figure 9: Source code

the proof of pred5. Invoke **Intro** command at the goal 0. Then we obtain the following:

```
|  pred5 (P::Prop) :: (Exist D (\(x::D) -> P)) <=> P
|    = struct
|        fst::Exist D (\(x::D) -> P) -> P
|          = {}0
|        snd::P -> Exist D (\(x::D) -> P)
|          = {}1
|
```

Let us work on the new goal 0. Invoke **Intro** command at the goal 0. Then we obtain the following:

```
|  pred5 (P::Prop) :: (Exist D (\(x::D) -> P)) <=> P
|    = struct
|        fst::Exist D (\(x::D) -> P) -> P
|          = \(h::Exist D (\(x::D) -> P)) ->
|            {}2
|        snd::P -> Exist D (\(x::D) -> P)
|          = {}1
|
```

The definition of `Exist` is the dependent sum, whose canonical objects are dependent pairs $< d, p >$ of $d : D$ and $p : (\lambda x : D.P)d$. The latter type of p

reduces to P, because $\lambda x : D.P$ is a constant function. Since the `h` represents a dependent pair, its second element `h.snd` works for the goal 0. Input `h.snd` and invoke **Refine** command: This gives us the following:

```
|  pred5 (P::Prop) :: (Exist D (\(x::D) -> P)) <=> P
|    = struct
|        fst::Exist D (\(x::D) -> P) -> P
|          = \(h::Exist D (\(x::D) -> P)) ->
|                h.snd
|        snd::P -> Exist D (\(x::D) -> P)
|          = \(h::P) ->
|            {}1
|
```

Now let us work on the goal 1: The definition body of `snd` corresponds to a proof of `P => Exist D (\(x::D)-> P)`. Invoke **Intro** command at the goal 1 twice. Then we obtain the following:

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

Let us work on the goal 1. We need an object of type `D`. Here we use the non-emptiness of the domain. Input `d0` into the goal 1 and invoke **Refine** command: This gives us the following:

```
snd ::P -> Exist D ( \(x::D)-> P)
  =  \(h::P)->
     struct
       fst ::D = d0
       snd ::P = {}2
```

Let us turn to the goal 2. Clearly `h` will do. Input `h` and invoke **Refine** command: This gives us the following:

```
pred5 (P::Prop) :: (Exist D (\(x::D)-> P)) <=> P
  = struct
    fst ::Exist D ( \(x::D)-> P)-> P
      = \(h::Exist D (\(x::D)-> P))->
        h.snd
    snd ::P -> Exist D ( \(x::D)-> P)
      =  \(h::P)->
         struct
           fst ::D = d0
           snd ::P = h
```

This completes the proof of `pred5`.

6.4.6 Exercises.

Prove the following using in `Agda`. (Only the left hand side is showed in each quiestion.)

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

```
Exer6 (P::Prop)(Q::Pred D) ::
  (Exist D (\(x::D)-> P && Q x))
   <=> (P && (Exist D Q))
```

2. $\exists x Px \equiv \exists y Py$. (Change of bound variables).

```
Exer7 (P::Pred D) ::
  (Exist D (\(x::D)-> P x)) <=> (Exist D (\(y::D)-> P y))
```

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

```
Exer8 (Q::Pred D) (P::Prop) ::
  (P || (Exist D Q)) <=>
  (Exist D (\(y::D)-> (P || (Q y)))
```

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

```
Exer9 (Q::Pred D)(P::Prop) ::
  (P || (Forall D Q)) =>
  (Forall D (\(y::D)-> (P || (Q y))
```

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

```
Exer10 (P::Pred D)(Q::Pred D) ::
  ((Forall D P) && (Forall D Q)) <=>
  Forall D (\(y::D) -> ((P y) && (Q y)))
```

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

```
Exer11 (P::Pred D)(Q:: Pred D) ::
  ((Forall D P) || (Forall D Q)) =>
  Forall D (\(y::D) -> ((P y) || (Q y)))
```

6.4.7 Classical first-order predicate logic

Let P be a predicate without free occurence of variable x. We shall prove a proposition

$$\forall x(P \vee Q(x)) \supset (P \vee \forall x Q(x)).$$

We need law of exclusive middle to prove it as we shall see later.

Let us begin with a new package `ClassicalPred`.

```
|--#include "LogicLib.agda"
|
|package ClassicalPred(D::Set)(d0::D) where
|
|  open LogicLib use Prop, Pred, (||), inl, inr, Not, Forall, (=>)
|
|  postulate em ::(X::Prop)-> (X || Not X)
|
|  pred6 (P::Prop)(Q::Pred D) ::
|   Forall D (\(x::D)-> (P || Q x)) => (P || (Forall D Q))
|   = {}0
|
```

The complete proof of the proposition in Agda is in Figure 10. Let us work on

```
--#include "LogicLib.agda"

package ClassicalPred(D::Set)(d0::D) where

  open LogicLib use Prop, Pred, (||), inl, inr, Not, Forall, (=>)

  postulate em ::(X::Prop)-> (X || Not X)

  pred6 (P::Prop)(Q::Pred D) ::
   (Forall D (\(x::D)-> P || Q x)) => (P || (Forall D Q))
   = \(h::Forall D (\(x::D)-> 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 10: Source code

the goal 0. Invoke **Intro** command at the goal 0, and we obtain the following:

```
pred6 (P::Prop)(Q::Pred D)::
 (Forall D (\(x::D)-> (P || Q x)) => (P || (Forall D Q))
 =  \(h::Forall D (\(x::D)-> (P || (Q x)))->
    {}1
```

where the subgoal is of the following type.

```
?1 :: P || Forall D Q
```

Let us give an informal proof of the proposition at first. We need to prove $P \vee \forall x Q(x)$ 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 $P \vee \forall x Q(x)$. If P does not hold in the model, then $\forall x Q(x)$ must hold since $\forall x(P \vee Q(x))$ holds. Thus we again obtain $P \vee \forall x Q(x)$.

Now we encode this proof in `Agda`. We need the law of excluded middle for P. Input `em P` in the goal 1 and invoke **Case** command `C-c,C-c`, and we obtain the following:

```
pred6 (P::Prop)(Q::Pred D) ::
 Forall D (\(x::D)-> (P || Q x)) => (P || (Forall D Q))
 = \(h::Forall D (\(x::D)-> P || (Q x)))->
   case em P of
      (inl x)->
        {}2
      (inr y)->
        {}3
```

where the subgoal is of the following type.

```
?3 :: P || (Forall D Q)
?2 :: P || (Forall D Q)
```

The object `em P` represents a proof of `P||(Not P)`. Instead of `x` and `y`, we shall use better names: Edit them to `p` and `np`, respectively. Then `p` and `np` have types `P` and `Not P`.

```
pred6 (P::Prop)(Q::Pred D) ::
 Forall D (\(x::D)-> (P || Q x)) => (P || (Forall D Q))
 = \(h::Forall D (\(x::D)-> P || (Q x)))->
    case em P of
      (inl p)->
        {}2
      (inr np)->
        {}3
```

Let us work on the goal 2. In this case, Input `inl p` and invoke **Refine** command. Then we obtain the following:

```
pred6 (P::Prop)(Q::Pred D) ::
 (Forall D (\(x::D)-> P || Q x) => (P || (Forall D Q))
 = \(h::Forall D (\(x::D)-> P || (Q x)))->
    case em P of
      (inl p)->
        inl p
      (inr np)->
        {}3
```

Let us work on the goal 3. In this case, we need a lemma claiming $\forall x Q x$ under the assumption of $\neg P$.

Input `lem` into the goal 3 and invoke **Let** command `C-c, C-l`: Then we obtain the following

```
pred6 (P::Prop)(Q::Pred D) ::
 Forall D (\(x::D)-> P || Q x) => (P || (Forall D Q))
 = \(h::Forall D (\(x::D)-> P || (Q x)))->
```

```
case em P of
(inl p)-> inl p
(inr np)->
  let lem::{}1
        = {}2
  in  {}3
```

where the subgoal is of the following type.

```
?3 :: P || Forall D Q
?2 :: ?1
?1 Type
```

We want the lemma for each x of the domain D. Hence Edit the let expression as follows and invoke **chase-load** command: Then we obtain the following:

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

Invoking **chase-load** command is to add a newly defined local variable `x` to the context list[21] kept internally in `Agda`. Input `Q x` into the goal 1 and invoke **Refine** command: Then we obtain the following:

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

Let's analyse the type `Forall D (\(x::D)-> P || (Q x))`. By the definition of `Forall`, it reduces to `(x::D) -> P || (Q x)`, which is the type of `h`. Hence `h x`, with `x` of type `D`, has the type `P || (Q x)`.

Now we argue by cases of `h x` whether it proves `P` or `Q x`. Input `h x` into the goal 2 and invoke **Case** command:

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

Let us work on the goal 4. Here we use absurd elimination: `x'` and `np` have types `P` and `Not P`. Hence `np x'` has type `Absurd`. Input `np x'` into the goal 4 and invoke **Case** command, which completes the object:

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

[21] The context list can be seen by the command **Context** `C-c |`. (See Section 5.2.3.)

Work on the goal 5. The variable `y` has type `Q x`. Input `y` into the goal 5, and invoke **Refine** command: This completes the object.

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

Since `lem` has type `(x::D)-> Q x`, or equivalently type `Forall D Q`, `inr lem` does `P || Forall D Q`. Input `inr lem` into the goal 3 and invoke **Refine** command.

```
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.

6.4.8 Exercises.

Prove the following using in `Agda`.

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

```
Exer12 (P::Prop)(Q::Pred D) ::
  (Forall D (\(x::D)-> P || Q x)) =>
  (P || (Forall D Q))
```

6.4.9 First-order arithmetic

In this subsection, we shall prove the associativity of addition in the first-order arithmetic by using mathematical induction.

In `IntroLib.agda`, the set `Nat` of natural numbers are defined:

```
data Nat = zer | suc (m::Nat)
```

Now let us examine the addition `add` on the natural numbers defined in `IntroLib.agda`:

```
add (m::Nat) (n::Nat) :: Nat
    = case m of
        (zer) -> n
        (suc m') -> suc@_ (add m' n)

(+)(m::Nat)(n::Nat) :: Nat = add m n
```

The addition '+' is an infix operator, 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 `Id Nat`.

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

```
--#include "IntroLib.agda"
--#include "LogicLib.agda"

package NatAssoc where

  open IntroLib Nat, zer, suc, add, (+)
  open LogicLib use Id, mapId
  ass_add' (x,y,z::Nat) ::
     Id Nat (x + y + z) (x + (y + z))
     = {}0
```

The complete proof of the proposition in `Agda` is in Figure 11.

```
--#include "IntroLib.agda"
--#include "LogicLib.agda"

package NatAssoc where

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

Figure 11: Source code

Precedence of the operators. There is a fixed set of operator precedence in `Agda` (refer to Coquand [2]): Since the operator `+` is always implemented to be left-associative, we may use the expression `x + y + z` instead of `(x + y) + z`.

Let us prove $(x + y + z) = (x + (y + z))$ for all $x, y, z : Nat$. We argue by induction on x.

Base case. The expressions on both sides reduce to $y + z$. Done.

Inductive case. We have the following equality:

$$\begin{array}{rcll} suc(x') + y + z & = & suc(x' + y + z) & \text{By the def. of } (+). \\ & = & suc(x' + (y + z)) & \text{By I.H..} \\ & = & suc(x') + (y + z) & \text{By the def. of } (+). \end{array}$$

Now let us construct the above proof in `Agda`. We generally discover functions to be used in a definition in due course. It is hard to predict them in advance. In this case, `mapId` is the one.

Remember that the addition + is defined on the first parameter. We shall construct the definition body of ass_add' inductively on x. Input x into the goal 0 and invoke **Case** command. Then we obtain the following:

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

where the subgoals 1 and 2 have the following type.

```
?2 :: Id Nat (suc@_ m + y + z) (suc@_ m + (y + z))
?1 :: Id Nat (zer + y + z) (zer + (y + z))
```

Exchange the variable m to x' in the source code and look at the goal 1. What is a proof object of this type? The expressions zer@_ + y + z and zer@_ + (y + z) are both evaluated to the same value y + z. Thus the type of the goal 1 is reduced to Id Nat (y + z) (y + z). Obviously the object of this type is ref@_ (y + z).

Input ref@_ (y + z) into the goal 1, and invoke **Refine** command.

```
ass_add'(x,y,z::Nat) ::
   Id Nat (x + y + z) (x + (y + z))
   = case x of
     (zer  )-> ref@_ (y + z)
     (suc x')-> {}2
```

Now turn to the goal 2: It has the type

```
Id Nat (suc@_ x' + y + z) (suc@_ x' + (y + z))
```

Hence the type is evaluated to the following:

```
Id Nat (suc@_ (x' + y + z)) (suc@_ (x' + (y + z)))
```

Our argument will be as follows: Note that the expression (ass_add' x' y z) has the type

```
Id Nat (x' + y + z) (x' + (y + z))
```

It means that the proof object (ass_add' x' y z) represents the induction hypothesis, claiming the associativity of '+' with respect to x', y and z. Now we use the mapId function:

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

Input mapId into the goal 2, and invoke **Refine** command.

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

```
mapId {}3
      Nat
      {}4
      {}5
      {}6
      {}7
```

From the type of `mapId` function, it is not difficult to figure out that the goals 3 and 4 are `Nat` and `suc :: Nat -> Nat`. Input `Nat` into the goal 3, and invoke **Refine** command. Input `suc` into the goal 4, and invoke **Refine** command.

```
ass_add' (x,y,z::Nat) ::
   Id Nat (x + y + z) (x + (y + z))
   = case x of
     (zer  )->
       ref@_ (y + z)
     (suc x')-> mapId Nat
                      Nat
                      suc
                      (add (add x' y) z)
                      (add x' (add y z))
                      {}7
```

where the subgoals 7 has the following type.

```
?7 :: Id Nat (add (add x' y) z) (add x' (add y z))
```

Input `ass_add'` into the goal 7, and invoke **Refine** command.

```
ass_add' (x,y,z::Nat) ::
   Id Nat (x + y + z) (x + (y + z))
   = case x of
     (zer  )->
       ref@_ (y + z)
     (suc x')-> mapId Nat
                      Nat
                      suc
                      (add (add x' y) z)
                      (add x' (add y z))
                      (ass_add' {}8
                                {}9
                                {}10)
```

Input `x'` in the goal 8, and invoke **Refine** command. Then we obtain the following:

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

```
                    Nat
                    succ
                    (add (add x' y) z)
                    (add x' (add y z))
                    (ass_add' x'
                              y
                              z)
```

This completes the proof of `ass_add'`.

6.4.10 Exercises.

Prove the following using in `Agda`.

1. We want to check by `Agda` that two definitions of the factorial functions are equivalent. Fill the rest of the proof (Hint: Use `mapId`, `tranId` and some basic lemmas for arithmetic. The commutativities for addition and multiplication are needed to prove the base case. Are they sufficient for the whole proof?).

```
open IntroLib use Nat, zer, suc, (+), (*), add, mul
open LogicLib use Id, ref, refId, symId, tranId, substId, mapId

fact (n::Nat)::Nat = case n of
                        (zer  )-> suc zer
                        (suc n')-> n * fact n'

facti (n,p::Nat)::Nat = case n of
                            (zer  )-> p
                            (suc n')-> facti n' (p * suc n')

com_add (m,n::Nat)::Id Nat (m + n) (n + m)
    = ?

com_mul (m,n::Nat)::Id Nat (m * n) (n * m)
    = ?

Exer13 (n,p::Nat):: Id Nat (facti n p) (p * (fact n))
    = case n of
        (zer  )-> tranId Nat p (suc zer * p) (p * (suc zer))
                    (tranId Nat p (zer + p) (p + zer)
                        (ref p)
                        (com_add zer p))
                    (com_mul (suc zer) p)
        (suc n')-> ?
```

A List of commands

All `Agda` commands can be invoked by key operations, or by selecting items in menus. The commands which are 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 items in goal menu depend on the context.

Commands are classified to four categories roughly.

Necessary commands you must know.

Important commands used very often.

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

Not recommended. Unsophisticated, or complicated commands. A user does not need to know about commands in this category. They may be changed or removed in the future.

A.1 Agda menu

Chase-import

key: `C-c C-x C-i`

category: *not recommended*

Reads the current buffer and included files, but type-checks the current buffer only.

Chase-load

key: `C-c C-x RET`

category: *necessary*

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

Check termination

key: `C-c C-x C-t`

category: *often*

Runs termination check on the current buffer.

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.

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, if any.

Quit

key: `C-c C-q`
category: *necessary*

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

Restart

key: `C-c C-x C-c`
category: *often*

(Re-)initializes the type-checker.

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.

Show goals

key: `C-c C-x C-a`
category: *often*

Shows all goals in the current buffer.

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 a tactic. In invoking this command, one of tactics and a constraint must be chosen. But it should be 0 in current version. Constraints are indexed by numbers (Ref. Show constraint).

Submitting bug report

key:
category:

(not implemented)

Suggest

key: `C-c C-x C-s`
category: *not recommended*

Suggests suitable expressions.

Text state

key: `C-c '`
category: *necessary*

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

Undo

key: `C-c C-u`

category: *important*

Cancels the last `Agda` command.

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).

A.2 Goal commands.

Auto

key:

category:

Try the auto tactic.(not implemented)

Abstraction

key: `C-c C-a`

category: *often*

Makes a template of a function expression with a given formal parameter.

Case

key: `C-c C-c`

category: *often*

Makes a template of a case expression with a given formal parameter.

Compute to depth

key: `C-c C-x 2 +`

category: *not recommended*

Prompts an expression and computes it to given depth. 'Compute 1 depth' corresponds to evaluating the innermost subexpression of a given expression.

Compute to depth 100

key: `C-c C-x +`

category: *important*

Prompts an expression and computes it to depth 100. 'Compute 1 depth' corresponds to reducing the innermost subexpression of a given expression.

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.

Context

key: `C-c |`

category: *important*

Shows context (names already defined) of the goal.

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.

Give

key: `C-c C-g`

category: *often*

Substitute a given expression to the goal and typechecks.

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.

Infer type

key: `C-c :`

category: *important*

Prompts an expression and infers the type of it, under the current context.

Infer type(unfolded)

key: `C-c C-x :`

category: *often*

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

Intro

key: `C-c C-i`

category: *important*

Introduces an abstraction or a struct in a goal. Different with abstraction command, intro command gives formal parameters automatically.

Let

key: `C-c C-l`

category: *often*

Makes a template of a let expression with a given formal parameter.

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 a expresseion in a given package. 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 refine command fails. (In case a goal is filled with "`OpList.cat`", refine command works.)

Unfold one

key: `C-c +`

category: *not recommended*

Prompts an expression and unfolds it one step.

B Sample programs in New Syntax

B.1 Intro.agda (Section 3)

```
----  intro.agda  ----
-- Section 3.1
data Bool' = true | false
data Nat = zer | suc (m::Nat)

-- Section 3.2
zero::Nat = zer
one::Nat = suc zer

succ (!m::Nat)::Nat = suc m
plus2 (!n::Nat) ::Nat = suc (suc n)
three::Nat = plus2 one

-- Invoke here 'Compute' Command
--foo::Nat = {! !}

-- Section 3.3
flip (!x::Bool')::Bool' =
    case x of
    (true )-> false
    (false)-> true

isZer (!n::Nat)::Bool' =
    case n of
    (zer   )-> true
    (suc n')-> false

-- Invoke here 'Compute Command'
--foo':: {! !} = {! !}

-- Section 3.4
add (!m::Nat) (!n::Nat)::Nat =
    case m of
        (zer   )-> n
        (suc m')-> suc (add m' n)

mul (!m::Nat) (!n::Nat)::Nat =
    case m of
        (zer   )-> zer
        (suc m')-> add n (mul m' n)

-- Invoke here 'Compute' Command
-- foo''::{! !} = {! !}

nonT (!m::Nat)::Nat
    = nonT (suc m)

-- Invoke here 'Compute' Command
-- foo'''::{! !} = {! !}
```

```
mutual
    f (!n::Nat)::Nat =
        case n of
        (zer   )-> one
        (suc n')-> add (mul three (f n')) (g n)
    g (!n::Nat)::Nat =
        case n of
        (zer   )-> zero
        (suc n')-> add (f n') (mul three (g n'))

(==) (!m::Nat) (!n::Nat)::Bool' =
    case m of
        (zer   )->
            case n of
                (zer   )-> true
                (suc n')-> false
        (suc m')->
            case n of
                (zer   )-> false
                (suc n')-> m' == n'

(+) (!m::Nat) (!n::Nat)::Nat = add m n
(*) (!m::Nat) (!n::Nat)::Nat = mul m n

-- Section 3.5
flip'::Bool'->Bool' =
    \(x::Bool')->
        case x of
        (true )-> false
        (false)-> true

add'::Nat->Nat->Nat =
    \(m::Nat)->
    \(n::Nat)->
        case m of
        (zer   )-> n
        (suc m')-> suc (add' m' n)

twice::(Nat->Nat)->Nat->Nat =
     \(f::Nat -> Nat)->
     \(x::Nat)->
     f (f x)

-- Invoke here 'Compute' Command
--foo'''':{! !} = {! !}


-- Section 3.6
Tuple::Set = sig
            fst::Bool'
            snd::Nat

aPair::Tuple = struct
```

```
                fst::Bool' = true
                snd::Nat = suc zer

n::Nat = aPair.snd

-- Section 3.7
aNat :: Nat =
    let
        n::Nat = add three (add three one)
    in
        mul n n

-- Section 3.9
data List' (X::Set) = nil | con (x::X) (xs::List' X)

-- Invoke here 'Compute' Command
--foo2::{! !} = {! !}

length (!X::Set) (!xs::List' X)::Nat =
    case xs of
        (nil      )-> zer
        (con x xs')-> suc (length X xs')

-- Section 3.11
tail (!X::Set)::(xs::List' X)-> List' X =
     \(xs::List' X)->
     case xs of
     (nil      )->
       nil
     (con x xs')->
       xs'

-- Section 3.12

idata Vec (!A::Set) :: Nat -> Set where
    Zero :: Vec A zer
    Cons (n::Nat) (a::A) (v::Vec A n) :: Vec A (suc n)

idata LimitedNat :: Nat -> Nat -> Set where
    Lb (m,n::Nat) :: LimitedNat m n
    Suc (m,n::Nat) (x::LimitedNat m n) :: LimitedNat m (suc n)

-- Section 3.13
nil' (X::Set)::List' X =
    nil

con' (X::Set)::X -> (List' X) -> List' X =
    \(x::X) -> \(xs::List' X) -> con x xs
```

B.2 IntroLib.agda (Section 4)

```
----  Package derived from Intro.agda
```

```
package IntroLib where
    data Bool' = true | false
    data Nat = zer | suc (m::Nat)

    add (!m::Nat) (!n::Nat)::Nat =
        case m of
            (zer   )-> n
            (suc m')-> suc (add m' n)

    mul (!m::Nat) (!n::Nat)::Nat =
        case m of
            (zer   )-> zer
            (suc m')-> add n (mul m' n)

    (==) (!m::Nat) (!n::Nat)::Bool' =
        case m of
            (zer   )->
                case n of
                    (zer   )-> true
                    (suc n')-> false
            (suc m')->
                case n of
                    (zer   )-> false
                    (suc n')-> m' == n'

    (+) (!m::Nat) (!n::Nat)::Nat = add m n
    (*) (!m::Nat) (!n::Nat)::Nat = mul m n


    data List' (X::Set) = nil | con (x::X) (xs::List' X)

    length (!X::Set) (!xs::List' X)::Nat =
        case xs of
            (nil      )-> zer
            (con x xs')-> suc (length X xs')

    idata Vec (!A::Set) :: Nat->Set where
            Zero :: Vec A zer
            Cons (n::Nat) (a::A) (v::Vec A n) :: Vec A (suc n)
```

B.3 subList.agda(Section 5)

```
-- subList.agda

--#include "IntroLib.agda"
open Intro use List', nil, con

-- Preparing
cat (X::Set)::List' X -> List' X -> List' X =
    \(xs::List' X) -> \(ys::List' X) ->
```

```
        case xs of
          (nil      )-> ys
          (con x xs')-> con x (cat xs' ys)
----

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

subList (!X::Set)::(List' X)  -> (List' (List' X)) =
    \(ys::List' X) ->
        case ys of
          (nil     )->
            con nil nil
          (con x xs)->
            let zs ::  List' (List' X);
                zs  = subList X xs
            in  cat zs (addElem X x zs)
```

B.4 LogicLib.agda(Section 6)

```
-- LogicLib.agda

package LogicLib where

  -- Section 6.1.2
  Prop :: Type = Set

  (=>)(!X,!Y::Prop) :: Prop = X -> Y

  -- Section 6.1.4
  Pred (!X::Set) :: Type = X -> Prop

  -- Section 6.1.5
  Forall(!D::Set)(!P::Pred D) :: Prop = (x::D) -> P x

  -- Section 6.2.1
  (&&)(!X,!Y::Prop) :: Prop
    = sig{fst :: X; snd :: Y;}

  Pair (!X,!Y::Prop) (!x::X) (!y::Y):: X && Y
    = struct
        fst:: X = x
        snd:: Y = y

  Pair_fst (!X,!Y::Prop)(!xy:: X && Y) :: X
    = xy.fst

  Pair_snd (!X,!Y::Prop)(!xy:: X && Y) :: Y
```

```
  = xy.snd

-- Section 6.2.2
data (||)(!X,!Y::Prop)
  = inl (x::X) | inr (y::Y)

when (!X, !Y::Prop)(C:: (X || Y) -> Prop)
      (e:: (x::X) -> C (inl x))
      (f:: (y::Y) -> C (inr y))
      (p:: X || Y)
      :: C p
  = case p of
    (inl x)-> e x
    (inr y)-> f y

when' (!X,!Y,!Z::Prop)(!e::X => Z)(!f::Y => Z)
  :: (X || Y) => Z
  = \(p::X || Y)->
      case p of
      (inl x)-> e x
      (inr y)-> f y

-- Section  6.2.3
data Absurd =

-- Section 6.2.4
data Taut = tt

-- Section 6.2.5
Not (!X::Prop) :: Prop = X => Absurd

-- Section 6.2.6
Exist (!X::Set)(!Y::Pred X) :: Prop
  = sig{fst :: X; snd :: Y fst;}

dep_pair (!X::Set) (!Y::Pred X) (!x::X) (!y::Y x) :: Exist X Y
  = struct
      fst:: X = x
      snd:: Y fst = y

dep_fst (!X::Set) (!Y::Pred X) (!xy::Exist X Y) :: X
  = xy.fst

dep_snd (!X::Set) (!Y::Pred X) (!xy::Exist X Y) :: Y (dep_fst X Y xy)
  = xy.snd

split (!X::Set) (!Y::Pred X)
      (!Z::Exist X Y -> Prop)
      (!d:: (x::X) -> (y:: Y x) -> Z (dep_pair X Y x y))
      (!p::Exist X Y)
      :: Z p
  = d p.fst p.snd

split' (!X::Set) (!Y::Pred X) (!Z::Set)
```

```
      ::((x::X) -> Y x -> Z) -> Exist X Y -> Z
  = \(f::(x::X) -> (x'::Y x) -> Z) ->
      \(p::Exist X Y) -> f p.fst p.snd

-- Section 6.2.7
(<=>)(!X,!Y::Prop) :: Prop = (X => Y) && (Y => X)

-- Section 6.3.1
Rel (!X::Set) :: Type = X -> X -> Prop

-- Section 6.3.2
Reflexive (!X::Set)(!R::Rel X) :: Prop = (x::X) -> R x x

Symmetrical (!X::Set)(!R::Rel X) :: Prop
  = (x1::X) -> (x2::X) -> R x1 x2 -> R x2 x1

Transitive (!X::Set)(!R::Rel X) :: Prop
  = (x1::X) -> (x2::X) -> (x3::X) ->
    R x1 x2 -> R x2 x3 -> R x1 x3

-- Section 6.3.3
Substitutive (!X::Set)(!R::Rel X) :: Type
  = (P::Pred X) -> (x1::X) -> (x2::X) ->
    R x1 x2 -> P x1 -> P x2

-- Section 6.3.4
idata Id (!X::Set) :: X -> X -> Set where
  ref (x::X) :: Id X x x

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

tranId(!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

substId (!X::Set)::Substitutive X (Id X)
  = \(P::Pred X)-> \(x1::X)->
    \(x2::X)-> \(h::Id X x1 x2)->
    \(h'::P x1)-> case h of (ref x)-> h'

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)
```

References

[1] Nordström, B., Petersson, K. and Smith, J.M., *Programming in Martin-Löf's Type Theory*, available at
`http://www.cs.chalmers.se/Cs/Research/Logic/book/`.

[2] Nordström, B., Petersson, K. and Smith, J.M., *Martin-Löf's Type Theory*, pp. 1 - 37 in Handbook of Logic in Computer Science, vol. 5 (2000), Oxford Science Publication.

[3] Coquand, C., *Syntax* in *Agda documentation*,
`http://www.cs.chalmers.se/~catarina/agda/syntax.html`.

Index