Simple, Secure, Scalable Web Development with Opa

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

TABLE OF CONTENTS

Chapter 1

Introducing Opa

This is a great time to be a developer: As a community, we are changing the world, every day, thanks to the web, the cloud and their myriads of applications. Web applications and technologies available to developers have evolved tremendously since the first days of the web.

Opa is a new generation of web development platform that lets you write distributed web applications using a single technology. Throughout the pages of this book, we will introduce you to the many features of Opa. To quote but a few, Opa is concise, simple, concurrent, dynamically distributed, and secure.

1.1 What problem does Opa solve?

The web is evolving at a very fast pace. Unfortunately, the *process* of web application development has evolved neither quite as far, nor nearly as fast. Indeed, developing a web application in 2011 is not so different from developing a PC application in 1991: an impressive amount of developer time is spent not working on features or improvements but on *gluing* together antagonist technologies that were never designed to work together or in this context. In 1991, the technologies were the various hardware components, the low-level operating system, the high-level operating system and the memory itself, which could only be hacked into compliance through copious amounts of machine code. Per se, none of these technologies was wrong, but they just failed to fit together — a problem now known as *impedance mismatch*.

Today, the technologies involved include the various incompatible browsers, the server components or the database components, to quote but a few. As in 1991, none of these technologies is wrong, but they do not fit together. Because of this impedance mismatch, developers need to add to their application logic copious amounts of glue code just to get the components to agree on communication mechanisms and application-level protocols. Not only is this glue code tedious, repetitive and error-prone, but this is also where most safety and security issues arise.

In 1991, the answer to impedance mismatch was Software Development Kits, and indeed these SDKs proved very useful at allowing developers to develop anything at all, but failed to address the underlying issues. This was only achieved later, through better operating systems, better development tools, and resulted in today's Java Platform, C#, Objective-C or Python, for instance.

In 2011, web frameworks, both client-side and server-side, have the same role as the SDKs of the 1991s, with the same success but also the same shortcomings. But the problem remains: Building web applications is hackish, relies on glue code and is barely reusable.

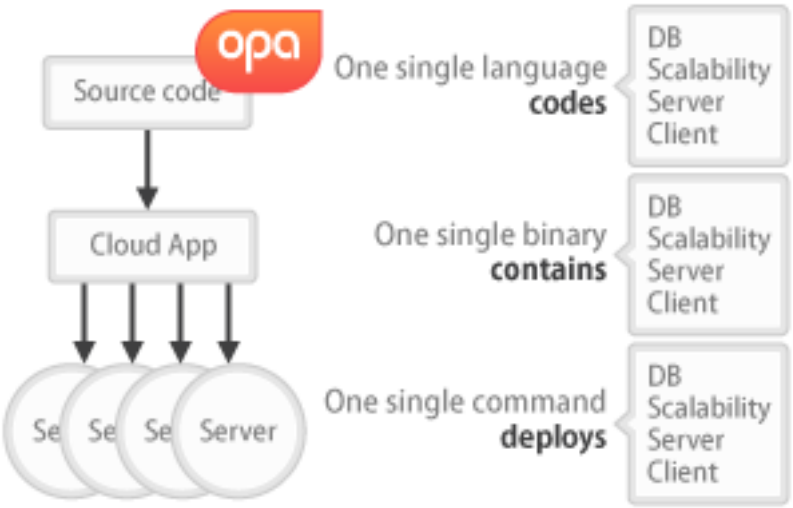
1.2 How Opa solves it

With Opa, we set out to provide another, better, way of developing web applications for the cloud. Opa is designed to get you to your finished app faster, concentrating only on the interesting parts, without the hassle of writing the glue or of using a programming language against its original design; also without having to keep such a constant watch on low-level security issues.

Opa is a programming language and a standard library comprising a database management system, a web server, a server-side framework, a client-side framework, a distribution middleware, a security audit tool, but without the complexity of deployment, administration, or impedance mismatch stemming from the use of many different technologies.

Developing with Opa is as simple as it gets: First, write your code with the Opa programming language — and only the Opa programming language, no JavaScript, no SQL, or anything such. Then compile the code, and let Opa analyze security, generate the code for the browser, for the server, for the database, and combine it with the web server. Finally, launch your executable and every browser, including mobile, can connect to your application. There is need to independently deploy, administer or maintain a database management system, a web server, frameworks or middleware. If you have lots of users, deploying in the cloud is just a command away.

Simply put, Opa is a programming language and its runtime, designed to carry web developers from start to finish of their development, from early prototype to seamless deployment.



1.3 What Opa is not

While Opa offers all the features that developers have come to expect from a high-end web server, a high-end distributed database management system, frameworks or distribution engine, Opa does *not* resemble Apache or Tomcat, MySQL or Oracle SQL, Drupal or Ruby on Rails, Google AppEngine or Hadoop. With some effort, Opa could certainly be used to mimic some of these components, but that is not how or why Opa was designed.

To this day, Opa does not offer a command-line database interface, or mechanisms to serve web pages from the file system. Opa is probably not well suited for writing embedded or desktop applications or anything else which does not run in the web.

1.4 What is it good for?

Opa was designed for all users, beginners or seasoned developers, students or professionals, but not for all applications.

Indeed, Opa was built from the bottom-up for *cloud* applications, and shines most with applications that require scalability, security, interactivity, real-time web and complex data manipulation.

Think of a social application. Numerous users communicating in real-time by chat, e-mail, forums or other means, through a rich user interface, but also through web services that can be accessed by third-party websites or native mobile interfaces. Perhaps these users can even modify shared appointment tables or work on some documents together. Developers need to combine complex information sources, extract the results quickly and get them back to the user in real-time. As the number of users grows, servers must be added, both to store large amounts of data, to manipulate it, and to respond quickly to end-users.

Out of the box, Opa features scalable storage, heavily concurrent/distributed manipulation of data, service-oriented API and load-balanced web serving. Even better, as social applications commonly require considerable computing power, the computing

performance of Opa ensures that fewer machines are required to handle crowds than would be required with other development solutions.

Or think of an e-Commerce store. Highly dynamic web pages, generated constantly from the database, and a user experience that may need to be altered quickly and without disrupting the flow, to match holiday seasons or ongoing ad campaigns, but also a base of orders that needs to be kept secure from would-be intruders, and safe from accidents, including hardware failures.

Opa is also great for these kinds of applications. Its content management engine is second to none, whether in flexibility, power or security. Opa's built-in security checks guarantee that the application will resist most attacks, while the replicated and versioned database ensures that you can always rollback to previous data or recover from outages.

We have used Opa to build social networks, e-Commerce applications, but also multiplayer web games, communication tools, developer-oriented tools — and of course the CMS behind our website.

1.5 A first peek at Opa

The rest of the book will introduce you in all details to Opa, but let us take a sneak peek preview and take a look at the simplest application: Hello, web.

```
server = one_page_server("Hello", -> <>Hello, web!</>)
```

This is the full source code of the application. Without any setup (besides the installation of the compiler), obtaining a complete, deployable application is only a matter of one step:

```
opa hello_web.opa
```

This command produces a stand-alone executable binary, containing everything required for execution, including database management, compiled requests, server code, client code, user interface and everything that may be needed to connect them or to handle malicious connections or inputs. Opa generated and checked all this code for you. You may now launch the server:

```
./hello_web.exe
```

Or, if you prefer, you may deploy it to a cloud:

```
opa-cloud hello_web.exe --host localhost --host my@my_server1 --host my@my_server2
```

Your application is launched, load-balanced, and servers share whichever information is necessary.

1.6 Summary

In the following chapters, we will introduce you to the various features and use-cases of Opa. Each chapter concentrates on writing one specific application, and on how best to achieve this using combinations of skills developed in previous and current chapter. At the end of the book, additional reference chapters recapitulate in detail all the concepts of the language and the platform.

This book assumes some knowledge of programming (any language should do) and web pages (knowledge of HTML and CSS will be useful). Any additional knowledge of web applications will let you understand better how much Opa makes development *right*.

1.7 Getting in touch

If you have any question or feedback, do not hesitate to contact us.

A few ways to get in touch:

- Opa mailing list;

- Stack Overflow, an excellent site for seeking help with programming problems (do not forget to mark Opa related questions with the “Opa” tag);
- through Twitter, as opalang.

We will be there!

Chapter 2

Getting Opa

This section is about installation and configuration of Opa. If you want to learn more about Opa first, you can safely skip this section and come back later.

At the time of this writing, Opa is available for MacOS X and Linux (both for 64 bits architectures; a 32 bits port is in progress). A Windows port is in the works and Opa should also work on a number of other operating systems (e.g. BSD) but these platforms are not supported for the moment.

2.1 Installing Opa

The easiest solution for installing Opa is to download an installer from our website.

2.1.1 MacOS X

- If you have not done so yet, install the XCode tools, provided by Apple with your copy of MacOS X. If you do not have XCode, you can either download the latest version from the Mac AppStore (paying) or download an older version from Apple's website (free, registration required). In addition to general development utilities, this package provides some of the low-level, Mac-specific, tools used by Opa to produce server-side executables.
- Download Opa for MacOS X.
- Once the download is complete, if your browser does not open automatically the file you have just downloaded, go to your Download folder and open this file. This should open a new Finder window containing the Opa installer package.
- Open the Opa installer package by double-clicking on it.
- Follow the instructions on the screen. You will need the password of an administrative account.
- Once the installation is complete, the Opa compiler will be installed in the directory

```
/opt/mlstate
```

and symbolic links will be created in `/usr/local/bin` hence you should not have to modify your PATH variable to enjoy the Opa compiler and its tools.

2.1.2 Ubuntu Linux, Debian Linux

The following instructions are also valid for all Debian-based Linux distributions.

- Download the Ubuntu Linux package.

- Once the download is complete, your operating system will offer you to install Opa using a tool such as `gdebi` or `kdebi`.
- Follow the instructions on the screen. You will need the password of an account with administrative rights.
- Once the installation is complete, the Opa compiler will be installed in the directory

```
/usr/bin
```

and the documentation and examples will be in

```
/usr/share/doc/opa
```

2.1.3 Arch Linux

A community package is in progress at AUR. It is not officially supported by us, and we have not tested it, but we are interested by any feedback about it.

2.1.4 Other Linux distribution

To install Opa on Suse, Red Hat, Fedora and other distributions of Linux which do not use the .deb system, or if you do not have administrative rights on your machine, take the following steps:

- Using your package manager, install packages:
  - `libssl-devel`
  - `zlib1g-devel`
- Or, if these packages do not exist in your distribution, install whichever packages provide
  - library `libssl.so`
  - library `libz.so`
- Download the Linux self-extracting package.
- When prompted by your browser, choose to save the file in a temporary directory, for instance, `/tmp`.
- To install as a user:
  - Set the execution permission on the downloaded program
  - Run it and follow the instructions on the screen
  - The compiler is installed by default in `~/mlstate-opa/bin`
- To install system-wide, open a terminal (if you are using KDE, the terminal is called Konsole)
  - In the terminal, write:

    ```
    sudo sh [complete path to the file you have just downloaded]
    ```

  - Follow the instructions on the screen
  - By default, the installation is done in `/usr/local`
- This installation comes with an un-installation script, in `[install prefix]/share/opa/uninstall.sh`

2.2 Building Opa from the sources

Should you wish to work on an unsupported platform, or to contribute to Opa, you will need to build Opa from sources. This requires a bit more work than the installation, but you should have no major difficulties. You will need:

- git (to download the source);
- ocaml 3.12.0-5 or later
- libgif 4.1 or later (dev version)
- libjpeg 8b-1 or later (dev version)
- libpng 1.2.44 or later (dev version)
- libssl 0.9.8 or later (dev version)
- libxft 2.2.0 or later (dev version)
- m4
- dblatex
- java 1.5 or later
- libx11 1.4.2 or later (dev version)
- zlib 1.2.3.4 or later (dev version)

In addition, if you are using MacOS X, you will need:

- GNU coreutils, part of MacPorts (free)
- The XCode suite, provided by Apple with your copy of MacOS X. If you do not have XCode, you can either download the latest version from the Mac AppStore (paying) or download an older version from Apple's website (free, registration required). This package provides some of the low-level, Mac-specific, tools used by Opa to produce server-side executables.

Once these dependencies are satisfied, take the following steps:

- Grab the sources from GitHub by entering in a terminal:

```
git clone git@github.com:MLstate/opalang.git
```

- In the same terminal, enter

```
cd opalang
./configure --prefix=SOME_DIRECTORY
make
make install
```

(You may need root privileges). This will install Opa in directory `SOME_DIRECTORY`

2.3 Setting up your editor

The package you installed provides two Opa modes, one for Emacs and one for Vim.

2.3.1 Emacs

On MacOS-X, either you're using Aquamacs and the package installation took care of it, or you should add the following line to your configuration file (which might be `~/.emacs`).

```
(autoload 'opa-mode "/Library/Application Support/Emacs/site-lisp/opa-mode/opa-mode.el" " ↩
    OPA editing mode." t)
(add-to-list 'auto-mode-alist '("\\.opa$" . opa-mode))
```

On Linux, add the following lines to your configuration file:

```
(autoload 'opa-mode "/usr/share/opa/emacs/opa-mode.el" "OPA editing mode." t)
(add-to-list 'auto-mode-alist '("\\.opa$" . opa-mode))
```

Tip
You may want to activate spell-checking on Opa comments and strings. To do so, type the command `M-x flyspell-prog-mode` within emacs.
And if you want this functionality activated each time you open an Opa file, you just need to add the following lines to your configuration file:

```
(defun enable_flyspell ()
  (ispell-change-dictionary "american")
  (flyspell-prog-mode)
)

;; Enable spell-checking on OPA comments and strings
(add-hook 'opa-mode-hook 'enable_flyspell)
```

2.3.2 Vim

If you are running Linux (resp. MacOS-X), copy files `/usr/share/opa/vim/{ftdetect,syntax}/opa.vim` (resp. `/opt/mlstate/share/opa/vim/{ftdetect,syntax}/opa.vim`) to your `.vim` directory, keeping the directory structure.

Tip
Instead of copying you can create a symbolic link. This will let you be automatically up-to-date with the latest mode every time you install a new version of Opa.

2.3.3 Eclipse

An experimental Eclipse plugin is available from GitHub. It is not fully functional, but it is good start, and we hope that the open source community can help us.

2.3.4 Other editors

Although we do not provide configuration files for other editors yet, we would be very happy to hear about it.

Chapter 3

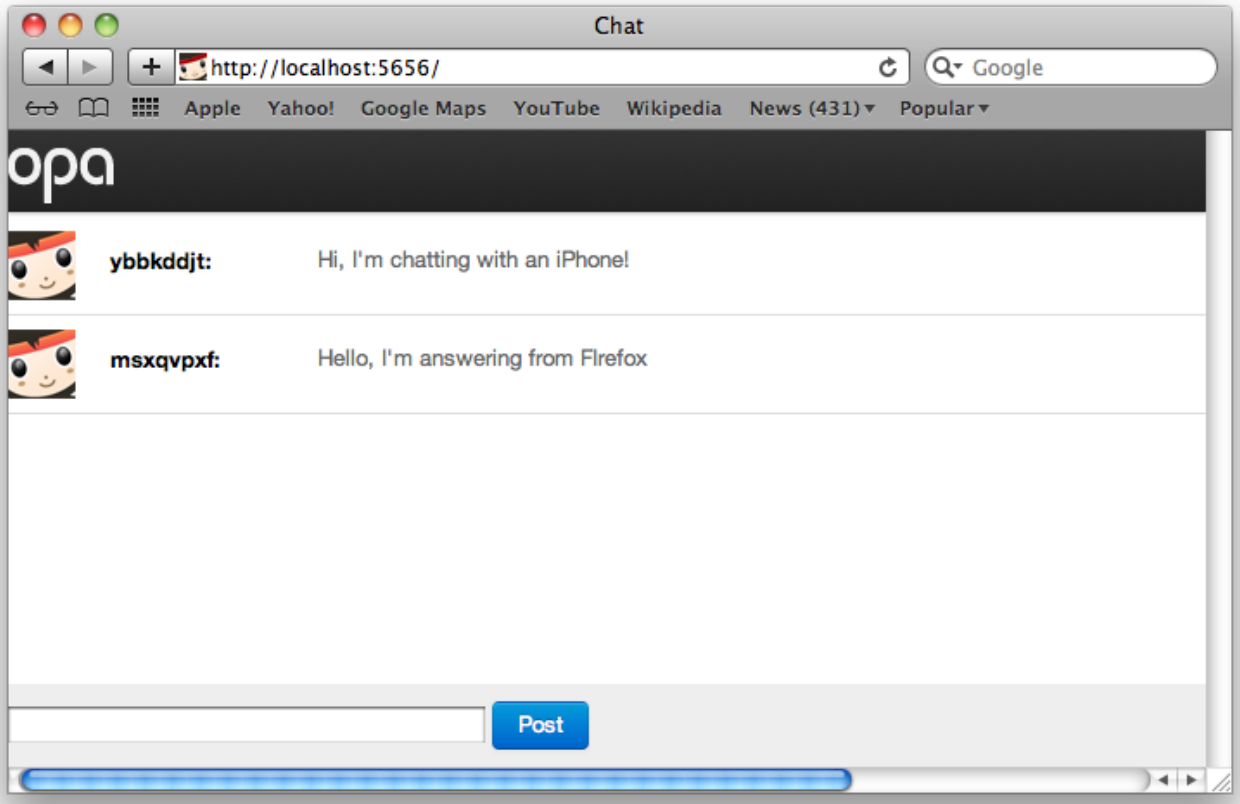
Hello, chat

Real-time web applications are notoriously difficult to develop. They require a complex and error-prone infrastructure to handle communications between client and server, and often from server to server, in addition to deep security checks against specific attacks that may prove quite subtle.

Opa makes real-time web simple. In this chapter, we will see how to program a complete web chat application in Opa — in only 20 lines of code, and without compromising security. Along the way, we will introduce the basic concepts of the Opa language, but also user interface manipulation, data structure manipulation, embedding of external resources, as well as the first building bricks of concurrency and distribution.

3.1 Overview

Let us start with a picture of the web chat we will develop in this chapter:



This web application offers one chatroom. Users connecting to the web application with their browser automatically join this chatroom and can immediately start discussing in real-time. On the picture, we have two users, using regular web browsers. For the sake of simplicity, in this application, we choose the name of users randomly.

If you are curious, the full source code of the application is listed at the end of this chapter. In the rest of the chapter, we will walk you through all the concepts and constructions: the communication infrastructure for the chatroom, the user interface, and finally, the main application.

3.2 Setting up communications

A chat is all about communicating messages between users. This means that we need to decide of what *type* of messages we wish to transmit, as follows:

```
type message = {author: string; text: string}
```

This extract determines that each `message` is composed of two informations: an `author` (which is a `string`, in other words, some text) and a `text` (also a `string`).

About types

Types are the shape of data manipulated by an application. Opa uses types to perform checks on your application, including sanity checks (e.g. you are not confusing a length and a color) and security checks (e.g. a malicious user is not attempting to insert a malicious program inside a web page or to trick the database into confusing informations). Opa also uses types to perform a number of optimizations.

In most cases, Opa can work even if you do not provide any type information, thanks to a mechanism of *type inference*. However, in this book, for documentation purposes, we will put types even in a few places where they are not needed.

We say that *type* `message` is a *record* with two *fields*, `author` and `text`. We will see in a few minutes how to manipulate a `message`.

At this stage, we have a complete (albeit quite useless) application. Should you wish to check that your code is correct, you can *compile* it easily. Save your code as a file `hello_chat.opa`, open a terminal and enter

Compiling Hello, Chat

```
opa hello_chat.opa
```

Opa will take a few seconds to analyze your application, check that everything is in order and produce an executable file. We do not really need that file yet, not until it actually does something. Opa will inform you that you have no *server* in your application — at this stage, your application is not really useful — but that is ok, we will add the *server* shortly.

So far, we have defined `message`. Now, it is time to use it for communications. For this purpose, we should define a *network*. Networks are a unit of communication between browsers or between servers. As you will see, communications are one of the many domains where Opa shines. To define one, let us write:

Networks
A network is a real-time web construction used to broadcast messages from one source to many observers. Networks are used not only for chats, but also for system event handling or for user interface event handling.
Networks themselves are built upon a unique and extremely powerful paradigm of *distributed session*, which we will detail in a further chapter.

```
room = Network.cloud("room"): Network.network(message)
```

This extract defines a *cloud network* called `room` and initially empty. As everything in Opa, this network has a type. The type of this network is `Network.network(message)`, marking that this is a network used to transmit informations with type `message`. We will see later a few other manners of creating networks for slightly different uses.

And that is it. With these two lines, we have set up our communication infrastructure — yes, really. We are now ready to add the user interface.

3.3 Defining the user interface

To define user interfaces, Opa uses a simple HTML-like notation for the structure, regular CSS for appearance and more Opa code for interactions. There are also a few higher-level constructions which we will introduce later, but HTML and CSS are more than sufficient for the following few chapters.

For starters, consider a possible skeleton for the user interface:

```
<div id=#conversation />
<input id=#entry />
<input type="button" value="Post" />
```

If you are familiar with HTML, you will recognize easily that this skeleton defines a few boxes (or `<div>`), with some names (or `id`), as well as a text input zone (or `<input>`) called `entry`. We will use these names to add interactions and style. If you are not familiar with HTML, it might be a good idea to grab a good HTML reference and check up the tags as you see them.

About HTML
There is not much more magic about HTML in Opa than the special syntax. For instance, the skeleton that we just defined is a regular Opa value, of type `xhtml`. You can for instance inspect its structure (with a `match` construct that we will se later), apply to it functions accepting type `xhtml`, or use it as the body of a function.

Actually, for convenience, and because it fits with the rest of the library, we will put this user interface inside a function, as follows:

```
start() =
(
  <div id=#conversation />
  <input id=#entry />
  <input type="button" value="Post" />
)
```

This extract defines a *function* called `start`. This function takes no *argument* and produces a HTML-like content. As everything in Opa, `start` has a type. Its type is `-> xhtml` .

About functions
Functions are bits of the program that represent a treatment that can be triggered as many times as needed (including zero). Functions that can have distinct behaviors, take *arguments* and all functions *produce* a *result*. Triggering the treatment is called *calling* or *invoking* the function.
Functions are used pervasively in Opa. A function with type `t1, t2, t3 -> u` takes 3 arguments, with respective types `t1`, `t2` and `t3` and produces a result with type `u`. A function with type `-> u` takes no arguments and produces a result with type `u`.
There are several ways of defining functions. You can either write `f(x1, x2, x3) = some_production` or `f = x1, x2, x3 -> some_production`, this is absolutely equivalent. Similarly, for a function with no argument, you can either write `f() = some_production` or `f = -> some_production`. To call a function `f` expecting three arguments, you will need to write `f(arg1, arg2, arg3)`. Similarly, for a function expecting no argument, you will write `f()`.

Function syntax
In `f(x1, x2, x3) = some_production`, as well as in `f(arg1, arg2, arg3)`, there is no space between `f` and `(`. Adding a space changes the meaning of the extract and would cause an error during compilation.

At this stage, we can already go a bit further and invent an author name, as follows:

```
start() =
(
  author = Random.string(8)
  <div id=#conversation />
  <input id=#entry />
  <input type="button" value="Post" />
)
```

This defines a value called `author`, with a name composed of 8 random characters.

With this, we have placed everything on screen and we have already taken a few additional steps. That is enough for the user interface for the moment, we should get started with interactivity.

3.4 Sending and receiving

We are developing a chat application, so we want the following interactions:

1. at start-up, the application should *join* the room;
2. whenever a message is broadcasted to the room, we should display it;
3. whenever the user presses return or clicks on the button, a message should be broadcasted to the room.

For these purposes, let us define a few auxiliary functions.

```
broadcast(author) =
(
   text    = Dom.get_value(#entry)
   message = {author=author text=text}
   do Network.broadcast(message, room)
   Dom.clear_value(#entry)
)
```

This defines a function `broadcast`, with one argument `author` and performing the following operations:

- read the text entered by the user inside the input called `entry`, call this text `text`;
- create a record with two fields `author` and `text`, in which the value of field `author` is `author` (the argument to the function) and the value field `text` is `text` (the value just read from the input), call this record `message`;
- call Opa's network broadcasting function to broadcast `message` to network `room`;
- clear the contents of the input.

As you can start to see, network-related functions are all prefixed by `Network.` and user-interface related functions are all prefixed by `Dom.`. Keep this in mind, this will come in handy whenever you develop with Opa. Also note that our record corresponds to type `message`, as defined earlier. Otherwise, the Opa compiler would complain that there is something suspicious: indeed, we have defined our network to propagate messages of type `message`, attempting to send a message that does not fit would be an error.

About Dom

If you are familiar with web applications, you certainly know about the DOM already. Otherwise, it is sufficient to know that DOM, or Document Object Model, denotes the manipulation of the contents of a web page once that page is displayed in the browser. In Opa, elements in the DOM have type `dom`. A standard way to access such an element is through the selection operator , as in `#entry` which selects the element of id `"entry"` (ids must be unique in the page). A element whose id is the value of `id` (so `id` must be of type `string`).

Speaking of types, it is generally a good idea to know the type of functions. Function `broadcast` has type `string -> void`, meaning that it takes an argument with type `string` and produces a value with type `void`. Also, writing `{author=author text=text}` is a bit painful, so we added a syntactic sugar for this. We could have written just as well:

```
broadcast(author: string): void =
(
   text    = Dom.get_value(#entry)
   message = {~author ~text}
   do Network.broadcast(message, room)
   Dom.clear_value(#entry)
)
```

About void

Type `void` is an alias for the empty record, i.e. the record with no fields. It is commonly used for functions whose result is uninformative, such as functions only producing side-effects or sending messages.

This takes care of sending a message to the network. Let us now define the symmetric function that should be called whenever the network propagates a message:

```
user_update(x: message) =
(
  line = <div>{x.author}: {x.text}</div>
  Dom.transform([#conversation +<- line ])
)
```

The role of this function is to display a message just received to the screen. This function first produces a few items of user interface, using the same HTML-like syntax as above, and calls these items `line`. It then calls the `Dom.transform` function to add the contents of `line` at the end of box `conversation` we have defined earlier.

If you look more closely at the HTML-like syntax, you may notice that the contents inside curly brackets are probably not HTML. Indeed, these curly brackets are called *inserts* and they mark the fact that we are inserting not *text* `"x.author"`, but a text representation of the *value* of `x.author`, i.e. the value of field `author` of record `x`.

About inserts

Opa provides *inserts* to insert expressions inside HTML, inside strings and in a few other places that we will introduce as we meet them.

This mechanism of inserts is used both to ensure that the correct information is displayed and to ensure that this information is sanitized if needs be. It is powerful, simple and extensible.

In the extract, we have also seen function `Dom.transform`, whose role is to perform a (possibly complex) transformation on the screen, here by inserting the contents of `line` at the end of `conversation`. Other operators than `+<-` can be used to provide insertion at start, replacement, etc. Actually, this function takes a *list* of transformations — something materialized in the syntax by the square brackets.

We are now ready to connect interactions.

3.5 Connecting interactions

Let us connect `broadcast` to our button and our input. This changes function `start` as follows:

```
start() =
(
  author = Random.string(8)
  <div id=#conversation />
  <input id=#entry onnewline={_ -> broadcast(author)} />
  <input type="button" onclick={_ -> broadcast(author)} value="Post" />
)
```

We have just added *event handlers* to `entry` and our button. Both call function `broadcast`, respectively when the user presses *return* on the text input and when the user clicks on the button. As you can notice, we find again the curly brackets.

About event handlers

An *event handler* is a function whose call is triggered not by the application but by the user herself. Typical event handlers react to user clicking (the event is called `click`), pressing *return* (event `newline`), moving the mouse (event `mousemove`) or the user loading the page (event `ready`).

Event handlers are always connected to HTML-like user interface elements. An event handler always has type `Dom.event -> void`.

You can find more informations about event handlers in the Opa API documentation by searching entry `Dom.event` .

We will add one last event handler to our interface, to effectively join the network when the user loads the page, as follows:

```
start() =
(
  author = Random.string(8)
  <div id=#conversation onready={_ -> Network.add_callback(user_update, room)} />
  <input id=#entry onnewline={_ -> broadcast(author)} />
  <input type="button" onclick={_ -> broadcast(author)} value="Post" />
)
```

This `onready` event handler is triggered when the page is (fully) loaded and connects function `user_update` to our network.

And that is it! The user interface is complete and connected to all features. Now, we just need to add the `server` and make things a little nicer.

About _
Opa has a special value name _, pronounced "*I don't care*". It is reserved for values or arguments that you are not going to use, to avoid clutter. You will frequently see it in event handlers, as it is relatively rare to need details on the event (such as the position of the mouse pointer), at least in this book.

3.6 Bundling, building, launching

Every Opa application needs a `server`, to determine what is accessible from the web, so let us define one:

About servers
In Opa, every web application is defined by one or more server. A server is an entry point for the web, which offers to users a set of resources, such as web pages, stylesheets, images, sounds, etc.

```
server = Server.one_page_bundle("Chat", [], [], start)
```

This extract defines a `server`, using `Server.one_page_bundle`, one of the many server constructors provided by Opa. This specific constructor takes four arguments:

- the name of the application — users see it as the title in a browser tab or window;
- a list of additional static files to provide, in particular images or stylesheets — more on this later, for the moment, an empty list will be sufficient;
- a list of style informations to apply — more on this later, for the moment, an empty list will be sufficient;
- a function used to initialize and display the user interface — here, our function `start`.

About server
The value `server` is special. It does not have to be defined, although an application without a `server` is generally not really useful, but if it is defined it must have type `service`. You can also define more than one `server` in one application, generally to handle distinct subdomains and/or paths. We will cover this later.
You can find more information on the definition of a `server` in the documentation of the standard library, by searching for `Server`.

Well, we officially have a complete application. Time to test it!

We have seen compilation already:

Compiling Hello, Chat

```
opa hello_chat.opa
```

Barring any error, Opa will inform you that compilation has succeeded and will produce a file called `hello_chat.exe`. This file contains everything you need, so you can now launch it, as follows:

Running Hello, Chat

```
./hello_chat.exe
```

Congratulations, your application is launched. Let us visit it.

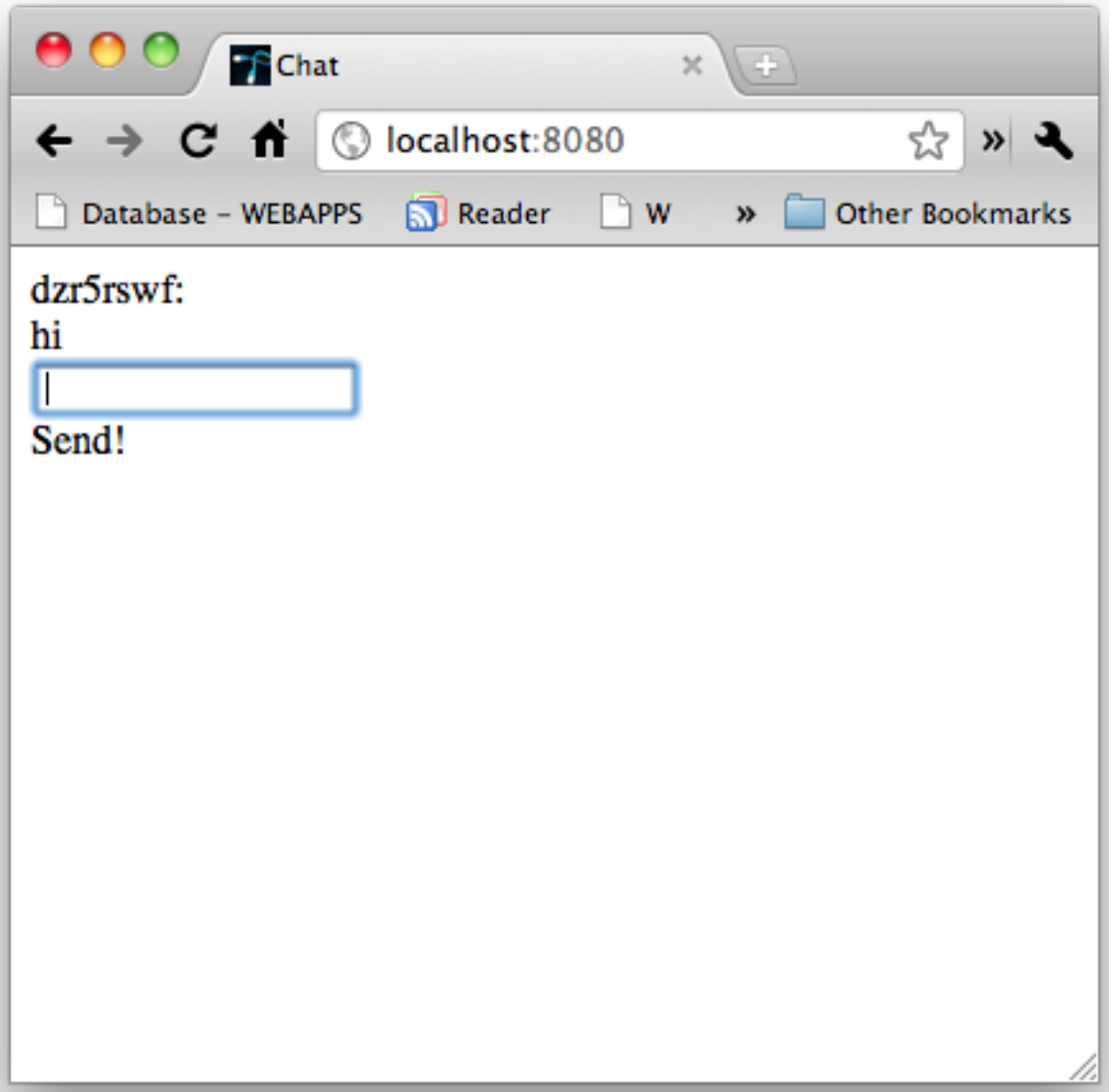
About hello_chat.exe
The opa compiler produces self-sufficient executable applications. The application contains everything is requires, including:

- webpages (HTML, CSS, JavaScript);
- any resource included with `@static_resource_directory`;
- the embedded web server itself;
- the distributed database management system;
- the initial content of the database;
- security checks added automatically by the compiler;
- the distributed communication infrastructure;
- the default configuration for the application;
- whatever is needed to get the various components to communicate.

In other words, to execute an application, you only need to launch this executable, without having to deploy, configure or otherwise administer any third-party component.

About 8080
By default, Opa applications are launched on port 8080. To launch them on a different port, use command-line argument `--port`. For some ports, you will need administrative rights.

As you can see, it works, but it is not very nice yet:



Perhaps it is time to add some style.

3.7 Adding some style

In Opa, all styling is done with stylesheets defined in the CSS language. This tutorial is not about CSS, so if you feel rusty, it is probably a good idea to keep a good reference at hand.

Of course, you will always need some custom CSS, specific to your application. Still, you can use some standard CSS to get you started with some predefined, nice-looking classes. Opa makes this as easy as a single import line:

```
import stdlib.themes.bootstrap
```

This automatically brings Bootstrap CSS from Twitter to your application, so you can use their predefined classes that will just look nice.

A first step is to rewrite some of our simple HTML stubs to give them more structure and add classes. The main user interface becomes (omitting the event handlers):

```
<div class="topbar"><div class="fill"><div class="container"><div id=#logo /></div></div></ ↩
    div>
<div id=#conversation class="container"></div>
<div id=#footer><div class="container">
  <input id=#entry class="xlarge"/>
  <div class="btn primary" >Post</div>
</div></div>
```

And the update function becomes:

```
user_update(x: message) =
(
  line =  <div class="row line">
            <div class="span1 columns userpic" />
            <div class="span2 columns user">{x.author}:</div>
            <div class="span13 columns message">{x.text}
            </div>
          </div>
  do Dom.transform([ #conversation +<- line ])
  Dom.scroll_to_bottom(#conversation)
)
```

Note that we have also added a call to `Dom.scroll_to_bottom`, in order to scroll to the bottom of the box, to ensure that the user can always read the most recent items of the conversation.

For custom style, you have two possibilities. You can either do it inside your Opa source file or as an external file. For this example, we will use an external file with the following contents:

```
/***Header***/
#logo {
  background: url("/resources/opa-logo.png") no-repeat scroll 0 0 transparent;
  height: 32px;
  margin: 10px 0 5px;
  width: 61px;
}

/***Conversation***/
#conversation {
  margin-top:58px;
  margin-bottom:40px;
  overflow:auto;
}
.line {
  border-bottom:1px solid #ddd;
  padding-bottom:8px;
  margin-bottom:8px !important;
}
.user, .message {padding-top:8px;}
.userpic {
  background: url("/resources/user.png") no-repeat 0 0;
  height: 40px;
  width:40px;
}
.user {color: #000;font-weight:bold;}
.message{color:#666;}

/***Footer***/
#footer {
  background:#eee;
```

```
  position:fixed;
  bottom:0;
  left:0;
  width:100%;
  padding:10px 0;
  text-align:left;
}
```

Create a directory called `resources` and save this file as `resources/css.css`. It might be a good idea to add a few images to the mix, matching the names given in this stylesheet (`opa-logo.png`, `user.png`) also in directory `resources`.

Now, we will want to instruct our `server` to access the directory and to use our stylesheet, by rewriting our `server` definition as follows:

```
server = Server.one_page_bundle("Chat",
   [@static_resource_directory("resources")], ["resources/css.css"], start)
```

In this extract, we have instructed the Opa *compiler* to embed the files of directory `resources`, offer them to the browser, and use our CSS file as the main style for the application.

About embedding
In Opa, the preferred way to handle external files is to *embed* them in the executable. This is faster, more secure and easier to deploy than accessing the file system.
To embed a directory, use *directive* `@static_resource_directory`.

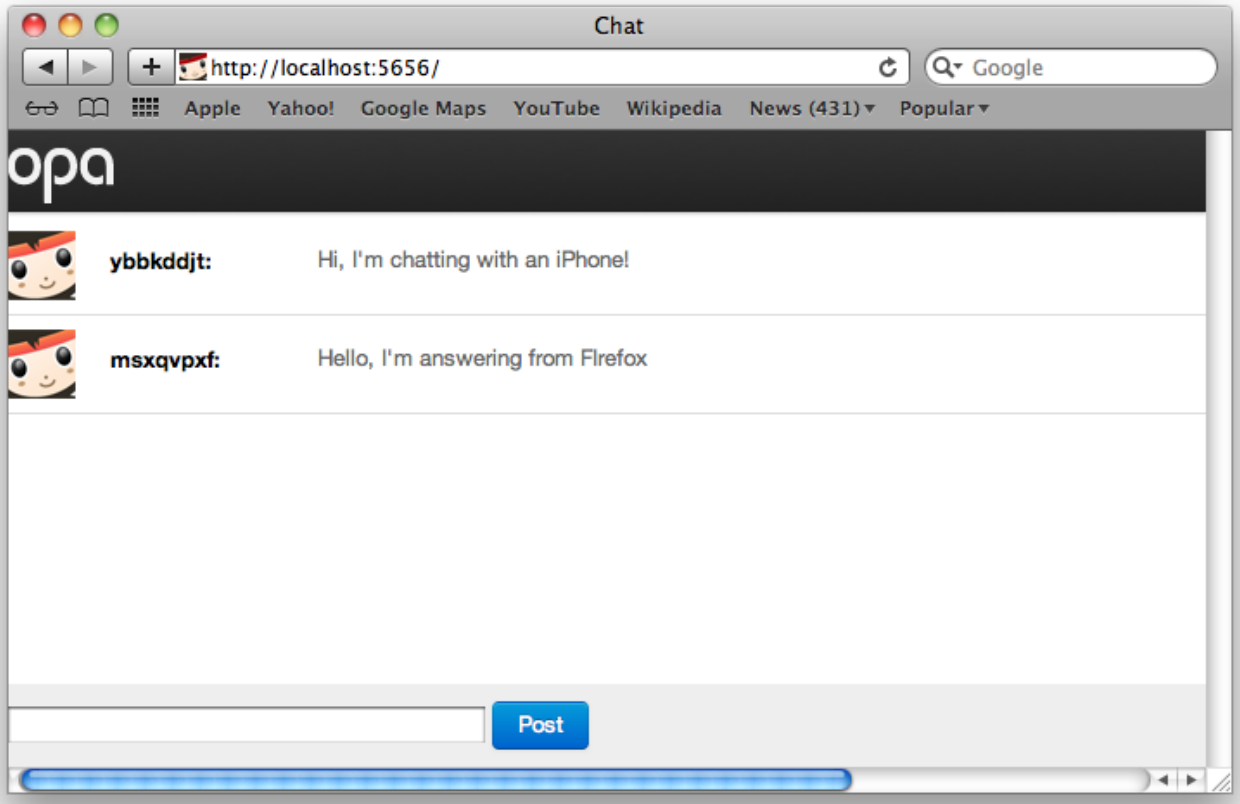
About directives
In Opa, a *directive* is an instruction given to the compiler. By opposition to *functions*, which are executed once the application is launched, directives are executed while the application is being built. Directives always start with character `@`.

While we are adding directives, let us take this opportunity to inform the compiler it does not have to protect the chatroom from clients, as follows:

```
@publish room = Network.cloud("room"): Network.network(message)
```

This will considerably improve the speed of the chat.

We are done, by the way. Not only is our application is now complete, it also looks nice:



As a summary, let us recapitulate the source file:

```
/**
 * {1 Import standard classes of bootstrap css}
 *
 * see http://twitter.github.com/bootstrap/
 */
import stdlib.themes.bootstrap

/**
 * {1 Network infrastructure}
 */

/**
 * The type of messages sent by a client to the chatroom
 */
type message = {author: string /**The name of the author (arbitrary string)*/
                ; text: string  /**Content entered by the user*/}

/**
 * The chatroom.
 */
@publish room = Network.cloud("room"): Network.network(message)

/**
 * {1 User interface}
 */

/**
 * Update the user interface in reaction to reception of a message.
```

```
 *
 * This function is meant to be registered with [room] as a callback.
 * Its sole role is to display the new message in [#conversation].
 *
 * @param x The message received from the chatroom
 */
user_update(x: message) =
  line = <div class="row line">
           <div class="span1 columns userpic" />
           <div class="span2 columns user">{x.author}:</div>
           <div class="span13 columns message">{x.text}
           </div>
         </div>
  do Dom.transform([#conversation +<- line ])
  Dom.scroll_to_bottom(#conversation)

/**
 * Broadcast text to the [room].
 *
 * Read the contents of [#entry], clear these contents and send the message to [room].
 *
 * @param author The name of the author. Will be included in the message broadcasted.
 */
broadcast(author) =
  text = Dom.get_value(#entry)
  message = {~author ~text}
  do Network.broadcast(message, room)
  Dom.clear_value(#entry)

/**
 * Build the user interface for a client.
 *
 * Pick a random author name which will be used throughout the chat.
 *
 * @return The user interface, ready to be sent by the server to the client on connection.
 */
start() =
  author = Random.string(8)
  <div class="topbar"><div class="fill"><div class="container"><div id=#logo /></div></div ↩
      ></div>
  <div id=#conversation class="container" onready={_ -> Network.add_callback(user_update, ↩
      room)}></div>
  <div id=#footer><div class="container">
    <input id=#entry class="xlarge" onnewline={_ -> broadcast(author)}/>
    <div class="btn primary" onclick={_ -> broadcast(author)}>Post</div>
  </div></div>

/**
 * {1 Application}
 */

/**
 * Main entry point.
 *
 * Construct an application called "Chat" (users will see the name in the title bar),
 * embedding statically the contents of directory "resources", using the global stylesheet
 * "resources/css.css" and the user interface defined in [start].
 */
server = Server.one_page_bundle("Chat",
        [@static_resource_directory("resources")],
        ["resources/css.css"], start)
```

Run

All this in 20 effective lines of code (without the CSS). Note that, in this final version, we have removed some needless parentheses, which were useful mostly for explanations, and documented the code.

3.8 Questions

3.8.1 *Where* is the `room`?

Good question: we have created a network called `room` and we haven't given any location information, so where exactly is it? On the server? On some client? In the database?

As `room` is shared between all users, it is, of course, on the server, but the best answer is that, generally, you do not need to know. Opa handles such concerns as deciding what goes to the server or to the client. We will see in a further chapter exactly how Opa has extracted this information from your source code.

3.8.2 Where are my headers?

If you are accustomed to web applications, you probably wonder about the absence of headers, to define for instance the title, favicon, stylesheets or html version. In Opa, all these concerns are handled at higher level. You have already seen one way of connecting a page to a stylesheet and giving it a title. As for deciding which html version to use, Opa handles this behind-the-scenes.

3.8.3 To `do` or not to `do`?

The source code presented above uses construction `do something`, which may be mysterious for you at this stage.

There is a very good reason for this construction: in Opa, every function definition (and more generally every value not at toplevel) ends with one *value*, which is the result of the function — conversely, once we have reached the first *value*, we have the result of the function, so the function is complete. And, in Opa, everything is a *value* except *value definitions* and *do actions*.

For instance, let us consider the following extract:

```
broadcast(author) =
(
   text    = Dom.get_value(#entry)
   message = {author=author text=text}
   do Network.broadcast(message, room)
   Dom.clear_value(#entry)
)
```

This is the definition of a value called `broadcast`. This value is a function whose production is the following value:

```
(
   text    = Dom.get_value(#entry)
   message = {author=author text=text}
   do Network.broadcast(message, room)
   Dom.clear_value(#entry)
)
```

or, equivalently,

```
text    = Dom.get_value(#entry)
message = {author=author text=text}
do Network.broadcast(message, room)
Dom.clear_value(#entry)
```

This value is composed of two value definitions, one `do` action and a simpler value, the result of the function.

If we had omitted this `do`, our function definition would have been understood as:

```
broadcast(author) =
(
   text    = Dom.get_value(#entry)
   message = {author=author text=text}
   Network.broadcast(message, room)   //This is the result

   Dom.clear_value(#entry)            //This is a syntax error
)
```

do vs. return
If you are familiar with languages using the `return` construction, you can think of `do` as the *opposite* of `return`. In Opa, we spend more time returning results than calling functions without using the result, so this is mirrored in the syntax.

The rule to remember is that you need a `do` if you are *computing* a value but you are not *defining* any value, and it is also not your result.

3.8.4 To `type` or not to `type`?

As mentioned earlier, Opa is designed so that, most of the time, you do not need to provide type information manually. However, in some cases, if you do not provide type information, the Opa compiler will raise a *value restriction error* and reject the code. Database definitions and value restricted definitions are the (only) cases in which you need to provide type information for reasons other than optimization, documentation or stronger checks.

For more information on the theoretical definition of a *value restriction error*, we invite you to consult the reference chapters of this book. For this chapter, it is sufficient to say that value restriction is both a safety and a security measure, that alerts you that there is not enough type information on a specific value to successfully guard this value against subtle misuses or subtle attacks. The Opa compiler detects this possible safety or security hole and rejects the code, until you provide more precise type information.

This only ever happens to toplevel values (i.e. values that are defined outside of any function), so sometimes, you will need to provide type information for such values. Since it is also a good documentation practice, this is not a real loss. If you look at the source code of Opa's standard library, you will notice that the Opa team strives to always provide such information, although it is often not necessary, for the sake of documentation.

3.9 Exercises

Time to see if this tutorial taught you something! Here are a few exercises that will have you expand and customize the web chat.

3.9.1 Customizing the display

Customize the chat so that

- the text box appears on top;
- each new message is added at the top, rather than at the bottom.

You will need to use operator `-<-` instead of `\<-+` to add at start instead of at end.

3.9.2 Saying "hello"

- Customize the chat so that, at startup, at the start of `#conversation`, it displays the following message to the current user:

```
Hello, you are user 8dh335
```

(replace `8dh335` by the value of `author`, of course).

- Customize the chat so that, at startup, it displays the following message to all users:

```
User 8dh335 has joined the room
```

- Combine both: customize the chat so that the user sees

```
Hello, you are user 8dh335
```

and other users see

```
User 8dh335 has joined the room
```

About comparison
To compare two values, use operator == or, equivalently, function `` `==` `` (with the backquotes). When comparing `x == y` (or `` `==`(x,y) ``), `x` and `y` must have the same type. The result of a comparison is a boolean. We write that the type of function == is `'a,'a -> bool`.

About booleans
In Opa, booleans are values `{true = void}` and `{false = void}`, or, more concisely but equivalently, `{true}` and `{false}`. You can check whether boolean `b` is true or false by using `if b then ... else ...` or, equivalently, `match b with {true} -> ... | {false} -> ...`.

3.9.3 Distinguishing messages between users

Customize the chat so that your messages are distinguished from messages by other users: your messages should be displayed with one icon and everybody else's messages should be displayed with the default icon.

3.9.4 User customization

- Let users choose their own user name.
- Let users choose their own icon. You can let them enter a URI, for instance.

More about xhtml
For security reasons, values with type `xhtml` cannot be transmitted from a client to another one. So you will have to find another way of sending one user's icon to all other users.

3.9.5 Security

As mentioned, values with type `xhtml` cannot be transmitted from a client to another one. Why?

3.9.6 And more

And now, an open exercise: turn this chat in the best chat application available on the web. Oh, and do not forget to show your app to the community!

Chapter 4

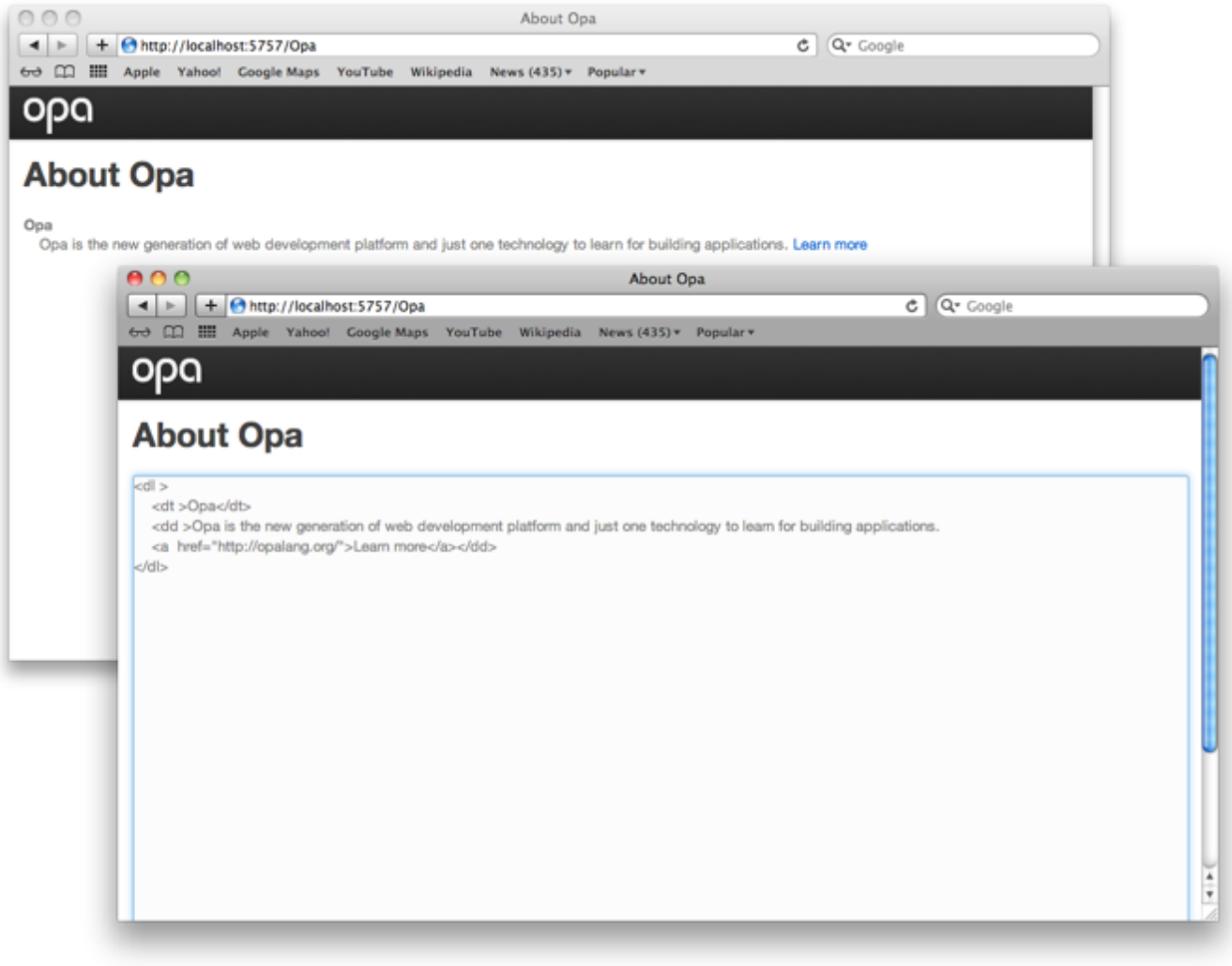
Hello, wiki

Wikis, and other form of user-editable pages, are components commonly encountered on the web. In itself, developing a simple wiki is error-prone but otherwise not very difficult. However, developing a rich wiki that both scales up and does not suffer from vulnerabilities caused by the content provided by users is quite harder. Once again, Opa makes it simple.

In this chapter, we will see how to program a complete, albeit simple, wiki application in Opa. Along the way, we will introduce the Opa database, the client-server security policy, the mechanism used to incorporate user-defined pages without breaking security, as well as more user interface manipulation.

4.1 Overview

Let us start with a picture of the wiki application we will develop in this chapter:



This web application stores pages and lets users edit them, using a HTML-like syntax that supports paragraphs, tables, etc.

If you are curious, this is the full source code of the application.

```
/**
 * {1 Import standard classes of bootstrap css}
 *
 * see http://twitter.github.com/bootstrap/
 */
import stdlib.themes.bootstrap

/**
 * {1 Import templates}
 */
import stdlib.web.template

/**
 * {1 Database and database interaction}
 */

/**
 * Contents of the wiki.
 *
 * Pages which do not exist have content "This page is empty".
 * Note: By definition, pages stored in the database are always well-formed.
 */
```

```
db /wiki: stringmap(Template.default_content)
db /wiki[_] = Template.text("This page is empty. Double-click to edit.")


/**
 * Read the content associated to a topic from the database and return the
 * corresponding source code.
 *
 * @param topic A topic (arbitrary string).
 * @return If a page has been saved in for [topic], the source code for this
 * page. Otherwise, the source code for the default page.
 */
@publish load_source(topic)   = Template.to_source(Template.default, /wiki[topic])

/**
 * Read the content associated to a topic from the database and return the
 * corresponding xhtml, ready to insert.
 *
 * @param topic A topic (arbitrary string).
 * @return If a page has been saved in for [topic], the xhtml for this
 * page. Otherwise, the xhtml for the default page.
 *
 * Note: This function does not perform any caching.
 */
@publish load_rendered(topic) = Template.to_xhtml(Template.default, /wiki[topic])

/**
 * Accept source code, save the corresponding document in the database.
 *
 * @param topic A topic (arbitrary string).
 * @param source Source code to store at this topic. If this source code
 * is syntactically valid, store the template datastructure
 * corresponding to its content [Template.content].
 * Otherwise, the source code is implicitly replaced by the document
 * representing this raw code and this document is saved in the database.
 *
 * @return In case of success, the xhtml for the page that has just been
 * saved. In case of failure, an error message.
 */
@publish save_source(topic, source) =
   match Template.try_parse(Template.default, source) with
    | ~{success}    ->
        do /wiki[topic] <- success;
        Template.to_xhtml(Template.default, success)
    | {failure = _} ->
        do /wiki[topic] <- Template.text(source);
        <>Error: {source}</>

/**
 * {1 User interface}
 */

/**
 * Set the user interface in edition mode.
 *
 * Load the source code for a topic, display an editable zone for this source code.
 *
 * @param topic The topic to edit.
 */
edit(topic) =
   do Dom.set_value(#edit_content, load_source(topic))
   do Dom.hide(#show_content)
```

```
   do Dom.show(#edit_content)
   do Dom.give_focus(#edit_content)
   void

/**
 * Set the user interface in reading mode.
 *
 * Save the source code for a topic (extracted from [#edit_content]),
 * display the rendered version.
 *
 * @param topic The topic to save.
 */
save(topic) =
   content = save_source(topic, Dom.get_value(#edit_content))
   do Dom.transform([#show_content <- content])
   do Dom.hide(#edit_content)
   do Dom.show(#show_content)
   void

/**
 * Main user interface
 *
 * @param topic The topic being consulted
 * @return A resource, ready to be passed to a dispatcher.
 */
display(topic) =
   Resource.styled_page("About {topic}", ["/resources/css.css"],
     <div class="topbar"><div class="fill"><div class="container"><div id=#logo></div></div ↩
         ></div></div>
     <div class="content container">
       <div class="page-header"><h1>About {topic}</></>
       <div class="well" id=#show_content ondblclick={_ -> edit(topic)}>{load_rendered( ↩
           topic)}</>
       <textarea rows="30" id=#edit_content onblur={_ -> save(topic)}></>
     </div>
   )

/**
 * {1 Main application}
 */

/**
 * Dispatch requests to the user interface
 *
 * Note: The empty request is dispatched as if it were "Hello".
 */
start =
   | {path = [] ... } ->
       display("Hello")
   | {~path ...}      ->
       display(String.capitalize(String.to_lower(String.concat("::", path))))

/**
 * Statically embed a bundle of resources
 */
server = Server.of_bundle([@static_include_directory("resources")])

/**
 * Launch the [start] dispatcher
 */
server = Server.simple_dispatch(start)
```

Run

In this listing, we define a database for storing the content of the pages in a safe format, we define the user interface and finally, the main application. In the rest of the chapter, we will walk you through all the concepts and constructions introduced.

As for the chat, we use Bootstrap CSS from Twitter with a single import line.

4.2 Setting up storage

A wiki is all about modifying, storing and displaying pages. This means that we need to set up some storage to receive these pages, as follows:

```
db /wiki: stringmap(Template.default_content)
```

This defines a *database path*, i.e. a place where we can store information, and specifies the type of information that will be stored there. Here, we use type `stringmap(Template.default_content)`. This is a `stringmap` (i.e. an association from `string` keys to values) containing values of type `Template.default_content`.

About database paths
In Opa, the database is composed of *paths*. A path is a named slot where you can store exactly one value at a time. Of course, this value can be a list, or a map, for instance, both of which act as containers for several values.
As everything else in Opa, paths are typed, with the type of the value that can be stored. Paths can be defined for any Opa type, except types containing functions or real-time web constructions such as networks. To guard against subtle incompatibilities between successive versions of an application, the type must be given during the definition of the path.

In turn, `Template.default_content` is the default content format for templates, i.e. the default method of representing rich text that can be modified by the user.

About template
On the web, letting users write HTML code that can be seen at a later stage by other users is a very bad idea, as it opens the way for numerous forms of attacks, sometimes quite subtle and hard to detect. Opa's *templating* mechanism is a manner of representing rich text that can be provided by a user, stored in the database and displayed to other users, without compromising security.
Although this topic is not covered by this chapter, the templating mechanism is designed for full extensibility - indeed, our website is assembled using this mechanism, including our own on-line editor.

It is generally a good idea to associate a default value to each path, as this makes manipulation of data easier:

```
db /wiki[_] = Template.text("This page is empty. Double-click to edit.")
```

The square brackets `[_]` are a convention to specify that we are talking about the contents of a map and, more precisely, providing a default value. Here, the default value is `Template.text("This page is empty. Double-click to edit.")`, i.e. a simple text in our templating system. As expected, it has type `Template.default_content`.

With these two lines, the database is set. Any data written to the database will be kept persistent, versioned and with regular snapshots. Should you stop and restart your application, the data will be checked and made available. If the data has been corrupted or is somehow incompatible with your application, you will be informed.

4.3 Loading, parsing and writing

Reading data from or writing data to the database is essentially transparent. For clarity and performance, we may start by defining two loading functions. One will load some content from the database and present it as source code that the user can edit, while a second will load the same content and present it as `xhtml` that can be displayed immediately:

```
load_source(topic)   = Template.to_source(Template.default, /wiki[topic])
load_rendered(topic) = Template.to_xhtml( Template.default, /wiki[topic])
```

In this extract, `topic` is the topic we wish to display or edit — i.e. the name of the page. Both functions use `Template.default`, the default template engine. This engine uses as source language a safe subset of HTML and produces content with type `Template.default_content`. The content associated with `topic` can be found in the database at path `/wiki[topic]`. Once we have this content, depending on our needs, we convert it to source code, as a `string`, or to a `xhtml` data structure used for display.

For parsing and saving data, the logic is slightly more complicated:

```
save_source(topic, source) =
(
   match Template.try_parse(Template.default, source) with
    | {success = x} ->
    (
           do /wiki[topic] <- x
           Template.to_xhtml(Template.default, x)
    )
    | {failure = _} ->
    (
           do /wiki[topic] <- Template.text(source)
           <>Syntax error<br/>: {source}</>
    )
)
```

This function takes two arguments: a `topic`, with the same meaning as above, and a `source`, i.e. a `string` meant to be understood by our templating engine. The call to function `Template.try_parse` submits this `source` to the default templating engine, which may either accept it (if the syntax was correct) or reject it (otherwise). For this wiki, we do not try to fix the syntax automatically or to fallback to an approximate syntax.

Function `Template.try_parse` can produce two kinds of results: *success* results and *failure* results. More precisely, by definition, the value returned by this function is always either a record `{ success = something }` or `{ failure = something}`. We call this a *sum* of records, and it has type `{ success = ... } /{ failure = ...}` — we will fill in the dots later.

About sum types

A value has a *sum type* `t/u` if, depending on the execution path, it can have either values of type `t` or values of type `u` . A good example of sum type is booleans, which are defined as `{false}/{true}` . Another good example of sum type is the type `list` of linked lists, whose definition can be summed up as `{nil} / {hd: ...; tl: list}`.

Note that sum types are not limited to two cases. Advanced applications commonly manipulate sum types with ten cases or more.

As you can see, we apply a `match ... with ...` operation to this result. This operation is known as *pattern-matching*, and it lets us determine safely whether the result was a *success* or a *failure*.

About pattern-matching

The operation used to branch according to the case of a sum type is called *pattern-matching*. A good example of pattern-matching is `if ... then ... else ...` . The more general syntax for pattern matching is `match ... with | case_1 -> ... | case_2 -> ... | case_3 -> ...`

The operation is actually more powerful than just determining which case of a sum type is used. Indeed, if we use the vocabulary of languages such as Java or C#, pattern-matching combines features of `if`, `switch`, `instanceof`/`is`, multiple assignment and dereferenciation, but without the possible safety issues of `instanceof`/`is` and with fewer chances of misuse than `switch`.

This pattern-matching has two cases. Let us concentrate on the first one:

```
    | {success = x} ->
    (
           do /wiki[topic] <- x
           Template.to_xhtml(Template.default, x)
    )
```

The items between the pipe (|) and the arrow (->) are known as the *pattern*. The corresponding case is executed if the *pattern matches* the value. Here, we have a pattern that accepts all records containing exactly one field called `success`. When our pattern-matching encounters such a record, it calls `x` the contents of field and executes everything that appears after the arrow (the *body*).

The first instruction writes `x` to the database at path `/wiki[topic]`. The second instruction produces a `xhtml`-formatted version of `x`, which is the result of our function.

Let us now examine the second case of our pattern-matching:

```
    | {failure = _} ->
    (
           do /wiki[topic] <- Template.text(source)
           <>Syntax error<br/>: {source}</>
    )
```

By definition, cases are executed in order. Here, the pattern matches all records containing exactly one field called `failure`. When our pattern-matching encounters such a record, it ignores the contents of the field — recall that _ is pronounced "I don't care" — and executes the body.

Here, we are in the error case, i.e. the user has entered a syntactically incorrect text. We could decide to perform sophisticated error reporting, but that goes beyond the scope of this chapter. We will rather employ an alternative strategy: we store the text entered by the user, but as source code, so that it can be modified at a later stage. For this purpose, we again call function `Template.text` to adopt the source code as immediate text, we write this to the database at path `/wiki[topic]` and then we produce a message stating that there was a syntax error.

With this, we have covered everything related to database, parsing or rendering. We can now move to the user interface.

4.4 User interface

As previously, we define a function to produce the user interface:

```
display(topic) =
(
   Resource.styled_page("About {topic}", ["/resources/css.css"],
     <div class="topbar"><div class="fill"><div class="container"><div id=#logo></div></div ↩
         ></div></div>
     <div class="content container">
       <div class="page-header"><h1>About {topic}</></>
       <div class="well" id=#show_content ondblclick={_ -> edit(topic)}>{load_rendered( ↩
           topic)}</>
       <textarea rows="30" id=#edit_content onblur={_ -> save(topic)}></>
     </div>
   )
)
```

This time, instead of producing a `xhtml` result, we have embedded this result in a `resource`, i.e. a representation for anything that the server can send to the client, whether it is a page, an image, or anything else. In practice, most applications produce a set of resources, as this is more powerful and more flexible than plain `xhtml`. Of course, you will need to use the appropriate `server` constructor, which we will introduce later.

A number of functions can be used to construct a `resource`. Here, we use `Resource.styled_page`, a function which constructs a web page, from its title (first argument), a list of stylesheets (second argument) and `xhtml` content. At this stage, the

`xhtml` content should not surprise you. We use `<div>` to display the contents of the page, or `<textarea>` (initially hidden) to modify them. When a user double-clicks on the content of the page (event `dblclick`), it triggers function `edit`, and when the user stops editing the source (event `blur`), it triggers function `save`.

Function `edit` is defined as follows:

```
edit(topic) =
(
   do Dom.set_value(#edit_content, load_source(topic))
   do Dom.hide(#show_content)
   do Dom.show(#edit_content)
   do Dom.give_focus(#edit_content)
   void
)
```

This function loads the source code associated with `topic`, sets the content of `#edit_content`, replaces the `<div>` with the `<textarea>`, gives focus to the `<textarea>` (purely for comfort) and returns `void`.

Similarly, function `save` is defined as follows, and shouldn't surprise you:

```
save(topic) =
(
   content = save_source(topic, Dom.get_value(#edit_content))
   do Dom.transform([#show_content <- content]);
   do Dom.hide(#edit_content);
   do Dom.show(#show_content);
   void
)
```

With these three functions, the user interface is ready. It is now time to work on the server.

4.5 Serving the pages

For this application, as we are serving several pages, we employ a new `server` constructor, which is more suited to our task: `Server.simple_dispatch`. This constructor also produces a `service`, but is suited to applications that need to return a number of distinct pages:

```
server = Server.simple_dispatch(start)
```

To get this to work, we need to define a function `start` as follows:

```
start(uri) = (
  match uri with
    | {path = {nil} ...} -> display("Hello")
    | {path = path  ...} -> display(String.concat("::", path))
)
```

This is another pattern-matching, built from some constructions that you have not seen yet. Pattern `{nil}` accepts the empty list. Pattern `...` accepts any number of fields in a record. In other words, the first case of our pattern-matching accepts any record containing at least one field named `path`, provided that this field contains exactly the empty list. In other words, this corresponds to a request for a URI with an empty path, such as "http://localhost:8080/".

The second pattern accepts any record containing at least one field named `path`. From the definition of pattern-matching, it is executed only if the first pattern did not match, for instance on a request to "http://localhost:8080/hello".

In both cases, we execute function `display`. The first case is trivial, while in the second case, we first convert our list to a string, with separator `"::"`.

Actually, we will make it a tad nicer by also ensuring that the first letter is uppercase, while the other letters are lowercase. Along the way, we can make this function shorter:

```
start =
   | {path = [] ...} -> display("Hello")
   | {~path     ...} -> display(String.capitalize(String.to_lower(String.concat("::", path) ↩
       )))
```

In this new version, we use a shorter syntax for pattern-matching. We use `[]` for the empty list — this is equivalent to `{nil}` — and we write `~path` — which is equivalent to `path = path`.

4.6 Adding some style

As in the previous chapter, without style, this example looks somewhat bland. As previously, we will fix this with an external stylesheet `resources/css.css`, with the following contents:

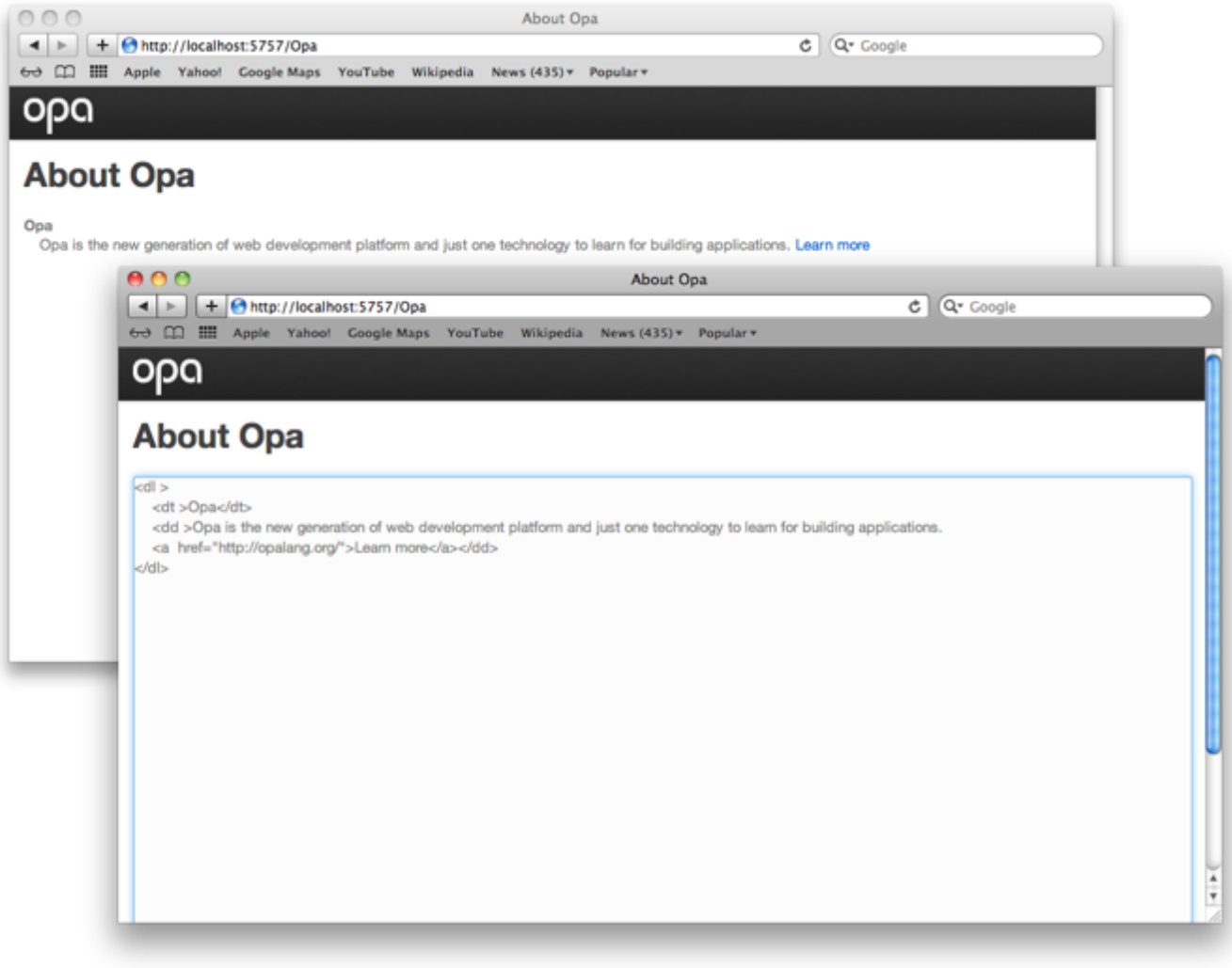
```
/***Header***/
#logo {
  background: url("/resources/opa-logo.png") no-repeat scroll 0 0 transparent;
  height: 32px;
  margin: 10px 0 5px;
  width: 61px;
}
/***Editing area***/
.content {margin-top:60px;}
#edit_content {
  width:100%;
  display: none; /* initially hidden */
}
```

One last step: including the stylesheet. For this purpose, instead of using a predefined function such as `Server.one_page_bundle`, we define a second `server`, just to include the contents of the directory, as follows:

```
server = Server.of_bundle([@static_include_directory("resources")])
```

Note that the order of `server` declarations is meaningful. This server must appear *before* our previous `server`.

With this final line, we have a complete, working wiki. With a few additional images, we obtain:



As a summary, let us recapitulate the source file:

```
/**
 * {1 Import standard classes of bootstrap css}
 *
 * see http://twitter.github.com/bootstrap/
 */
import stdlib.themes.bootstrap

/**
 * {1 Import templates}
 */
import stdlib.web.template

/**
 * {1 Database and database interaction}
 */

/**
 * Contents of the wiki.
 *
 * Pages which do not exist have content "This page is empty".
 * Note: By definition, pages stored in the database are always well-formed.
 */
db /wiki: stringmap(Template.default_content)
db /wiki[_] = Template.text("This page is empty. Double-click to edit.")
```

```
/**
 * Read the content associated to a topic from the database and return the
 * corresponding source code.
 *
 * @param topic A topic (arbitrary string).
 * @return If a page has been saved in for [topic], the source code for this
 * page. Otherwise, the source code for the default page.
 */
load_source(topic)   = Template.to_source(Template.default, /wiki[topic])

/**
 * Read the content associated to a topic from the database and return the
 * corresponding xhtml, ready to insert.
 *
 * @param topic A topic (arbitrary string).
 * @return If a page has been saved in for [topic], the xhtml for this
 * page. Otherwise, the xhtml for the default page.
 *
 * Note: This function does not perform any caching.
 */
load_rendered(topic) = Template.to_xhtml( Template.default, /wiki[topic])

/**
 * Accept source code, save the corresponding document in the database.
 *
 * @param topic A topic (arbitrary string).
 * @param source Source code to store at this topic. If this source code
 * is syntactically valid, store the template datastructure
 * corresponding to its content [Template.content].
 * Otherwise, the source code is implicitly replaced by the document
 * representing this raw code and this document is saved in the database.
 *
 * @return In case of success, the xhtml for the page that has just been
 * saved. In case of failure, an error message.
 */
save_source(topic, source) =
   match Template.try_parse(Template.default, source) with
    | ~{success}    ->
        do /wiki[topic] <- success;
        Template.to_xhtml(Template.default, success)
    | {failure = _} ->
        do /wiki[topic] <- Template.text(source);
        <>Error: {source}</>

/**
 * {1 User interface}
 */

/**
 * Set the user interface in edition mode.
 *
 * Load the source code for a topic, display an editable zone for this source code.
 *
 * @param topic The topic to edit.
 */
edit(topic) =
   do Dom.set_value(#edit_content, load_source(topic))
   do Dom.hide(#show_content)
   do Dom.show(#edit_content)
   do Dom.give_focus(#edit_content)
```

```
   void

/**
 * Set the user interface in reading mode.
 *
 * Save the source code for a topic (extracted from [#edit_content]),
 * display the rendered version.
 *
 * @param topic The topic to save.
 */
save(topic) =
   content = save_source(topic, Dom.get_value(#edit_content))
   do Dom.transform([#show_content <- content])
   do Dom.hide(#edit_content)
   do Dom.show(#show_content)
   void

/**
 * Main user interface
 *
 * @param topic The topic being consulted
 * @return A resource, ready to be passed to a dispatcher.
 */
display(topic) =
   Resource.styled_page("About {topic}", ["/resources/css.css"],
     <div class="topbar"><div class="fill"><div class="container"><div id=#logo></div></div ↩
         ></div></div>
     <div class="content container">
       <div class="page-header"><h1>About {topic}</></>
       <div class="well" id=#show_content ondblclick={_ -> edit(topic)}>{load_rendered( ↩
           topic)}</>
       <textarea clas="xxlarge" rows="30" id=#edit_content onblur={_ -> save(topic)}></>
     </div>
   )

/**
 * {1 Main application}
 */

/**
 * Dispatch requests to the user interface
 *
 * Note: The empty request is dispatched as if it were "Hello".
 */
start =
   | {path = [] ... } ->
       display("Hello")
   | {~path ...}      ->
       display(String.capitalize(String.to_lower(String.concat("::", path))))

/**
 * Statically embed a bundle of resources
 */
server = Server.of_bundle([@static_include_directory("resources")])

/**
 * Launch the [start] dispatcher
 */
server = Server.simple_dispatch(start)
```

Run

This is a total of 30 effective lines of code + CSS.

4.7 Questions

4.7.1 What about the wiki syntax?

For this tutorial chapter, we have used the default syntax provided by `Template`, which we have not detailed. That is because you already know this syntax: `Template` uses a subset of the syntax of HTML, without JavaScript. As mentioned earlier, this syntax can be extended. The general idea is to plug new *engines* into `Template` which can interpret new namespaces. We will discuss the specifics in another chapter, or you can already look at the documentation of module `Template`.

Opa also features a powerful mechanism for parsing simple or complex syntaxes, which you can easily use to replace the default syntax of `Template` with something closer to the usual Wiki syntax. We will introduce this mechanism in another chapter.

4.7.2 What about user security?

As mentioned, one of the difficulties when developing a rich wiki is ensuring that it has no security vulnerabilities. Indeed, as soon as a user may edit content that will be displayed on another user's browser, the risk exists of letting one user hide some JavaScript code (or possibly Flash or Java code) that will be executed by another user. This is a well-known technique for stealing identities.

You may attempt to reproduce this with the wiki, the chat, or any other Opa application. This will fail. Indeed, while lower-level web technologies make no difference between JavaScript code, text, or structured data, Opa does, and ensures that data that has been provided as the one can never be interpreted as one of the others.

Careful with the <script>
There is, actually, one exception to this otherwise bullet-proof guarantee: if a developer manually introduces `<script>` tag containing a insert, as follows, the possibility exists that a malicious user could take advantage of this to inject arbitrary code.

```
<script type="text/javascript">{security_hole}</script>
```

The bottom line is therefore: **do not introduce `<script>` tag containing an insert**. This is the *only* case that, at the time of this writing, Opa cannot check.

4.7.3 What about database security?

Now that we have a database, it is high time to think about *what* can be modified and *in which circumstances*. By default, Opa takes a conservative approach and ensures that malicious clients have access to as few entry points as possible — we call this *publishing* a function. By default, the only published functions are functions that users could trigger themselves by manipulation of the user interface, i.e. event handlers.

Published entry points
An *entry point* is a function that exists on the server but that could possibly be triggered by a user, possibly malicious.
Typically, every application contains at least one entry point, introduced by `server` — in the wiki, this is function `start`. Most applications also feature a manner for the client to send information to the server and trigger some treatment upon this information.
More generally, any function can be used as an entry point if it is *published*. By default, Opa will *automatically* publish event handlers.

Here, this means functions `edit` and `save`. In our listing, no other functions are *published*. Consequently, the `Template` mechanism is invoked only on the server. A direct consequence is that we can be certain that the `stringmap` is always well-formed and that `/wiki[x]` can only ever contain well-formed template content.

4.7.4 What about client-server performance?

At this stage, keeping the code in mind, you may start to wonder about performance and in particular about the number of requests involved in editing or saving a page. This is a very good point. Indeed, if your browser offers performance/request tracing tools, you will realize that calls to `edit` or `save` are quite costly.

This issue can be solved quite easily, but let us first detail the cost of saving:

1. The client sends a `save` request to the server (1 request).
2. The server prompts the client for the value of `edit_content` (2 requests).
3. The server instructs the client to hide `show_content` (2 requests).
4. The server instructs the client to show `edit_content` (2 requests).

Surely, this is not the best that Opa can do? Indeed, Opa can do better. To solve this, we only need to provide a little additional information to the compiler, namely to let it know that `load_source`, `load_rendered` and `save_source` have been designed to handle anything that can be thrown at them, and thus do not need to be kept hidden.

For this purpose, Opa offers a *publication* primitive: `@publish`. To apply it to our three functions, we simply add this directive where needed:

```
@publish load_source(topic)   = ...
@publish load_rendered(topic) = ...
@publish save_source(topic, source) = ...
```

And we are done!

With this simple modification, saving is now just one request. We will discuss `@publish` and the details of client-server *slicing* in a later chapter.

Once again, we may take a look at the complete application:

```
/**
 * {1 Import standard classes of bootstrap css}
 *
 * see http://twitter.github.com/bootstrap/
 */
import stdlib.themes.bootstrap

/**
 * {1 Import templates}
 */
import stdlib.web.template

/**
 * {1 Database and database interaction}
 */

/**
 * Contents of the wiki.
 *
 * Pages which do not exist have content "This page is empty".
 * Note: By definition, pages stored in the database are always well-formed.
 */
db /wiki: stringmap(Template.default_content)
db /wiki[_] = Template.text("This page is empty. Double-click to edit.")


/**
 * Read the content associated to a topic from the database and return the
 * corresponding source code.
```

```
 *
 * @param topic A topic (arbitrary string).
 * @return If a page has been saved in for [topic], the source code for this
 * page. Otherwise, the source code for the default page.
 */
@publish load_source(topic)   = Template.to_source(Template.default, /wiki[topic])

/**
 * Read the content associated to a topic from the database and return the
 * corresponding xhtml, ready to insert.
 *
 * @param topic A topic (arbitrary string).
 * @return If a page has been saved in for [topic], the xhtml for this
 * page. Otherwise, the xhtml for the default page.
 *
 * Note: This function does not perform any caching.
 */
@publish load_rendered(topic) = Template.to_xhtml(Template.default, /wiki[topic])

/**
 * Accept source code, save the corresponding document in the database.
 *
 * @param topic A topic (arbitrary string).
 * @param source Source code to store at this topic. If this source code
 * is syntactically valid, store the template datastructure
 * corresponding to its content [Template.content].
 * Otherwise, the source code is implicitly replaced by the document
 * representing this raw code and this document is saved in the database.
 *
 * @return In case of success, the xhtml for the page that has just been
 * saved. In case of failure, an error message.
 */
@publish save_source(topic, source) =
   match Template.try_parse(Template.default, source) with
    | ~{success}    ->
        do /wiki[topic] <- success;
        Template.to_xhtml(Template.default, success)
    | {failure = _} ->
        do /wiki[topic] <- Template.text(source);
        <>Error: {source}</>

/**
 * {1 User interface}
 */

/**
 * Set the user interface in edition mode.
 *
 * Load the source code for a topic, display an editable zone for this source code.
 *
 * @param topic The topic to edit.
 */
edit(topic) =
   do Dom.set_value(#edit_content, load_source(topic))
   do Dom.hide(#show_content)
   do Dom.show(#edit_content)
   do Dom.give_focus(#edit_content)
   void

/**
 * Set the user interface in reading mode.
 *
```

```
 * Save the source code for a topic (extracted from [#edit_content]),
 * display the rendered version.
 *
 * @param topic The topic to save.
 */
save(topic) =
   content = save_source(topic, Dom.get_value(#edit_content))
   do Dom.transform([#show_content <- content])
   do Dom.hide(#edit_content)
   do Dom.show(#show_content)
   void

/**
 * Main user interface
 *
 * @param topic The topic being consulted
 * @return A resource, ready to be passed to a dispatcher.
 */
display(topic) =
   Resource.styled_page("About {topic}", ["/resources/css.css"],
     <div class="topbar"><div class="fill"><div class="container"><div id=#logo></div></div ↩
         ></div></div>
     <div class="content container">
       <div class="page-header"><h1>About {topic}</></>
       <div class="well" id=#show_content ondblclick={_ -> edit(topic)}>{load_rendered( ↩
           topic)}</>
       <textarea rows="30" id=#edit_content onblur={_ -> save(topic)}></>
     </div>
   )

/**
 * {1 Main application}
 */

/**
 * Dispatch requests to the user interface
 *
 * Note: The empty request is dispatched as if it were "Hello".
 */
start =
   | {path = [] ... } ->
       display("Hello")
   | {~path ...}      ->
       display(String.capitalize(String.to_lower(String.concat("::", path))))

/**
 * Statically embed a bundle of resources
 */
server = Server.of_bundle([@static_include_directory("resources")])

/**
 * Launch the [start] dispatcher
 */
server = Server.simple_dispatch(start)
```

4.7.5 What about database performance?

Generally speaking, the Opa database is very fast. In particular, in our experience, a `stringmap` or an `intmap` is as fast as an *indexed* table in a relational database, sometimes much faster.

Of course, you do not have to restrict yourself to a `map` indexed by a `string` or an `int`. Indeed, any type that can be stored in the database can be used as an index. The chapter dedicated to the database will detail how you can do this and how to ensure that the performance of your database remains optimal.

4.8 Exercises

Time to put your new knowledge to the test.

4.8.1 Better handling of errors

When the user attempts to save content containing an error, instead of saving the result:

- keep editing;
- change the background color of `edit_content` to red;
- show a message informing that the user can simply reload the page to cancel changes.

You will need to use a CSS class for setting the background color. You may also wish to use the following functions:

```
Dom.add_class:    dom, /*class*/string -> void
Dom.remove_class: dom, /*class*/string -> void
```

4.8.2 Changing the default content

Customize the wiki so that the database contains not a `Template.default_content` but a `option(Template.default_con` i.e. a value that can be either `{none}` or `{some = x}`, where `x` is a `Template.default_content`.

Use this change to ensure that the default content of a page with topic *topic* is "We have no idea about *topic*. Could you please enter some information?".

4.8.3 Inform users of changes

Drawing inspiration from the chat, add an on-screen zone to inform users of changes that take place while they are connected.

4.8.4 Template chat

Modify your "Hello, Chat" application so that users can enter rich text, not just raw text.

4.8.5 Chat log

Modify your "Hello, Chat" application to add the following features:

- store the conversation as it takes place;
- when a new user connects, display the log of the conversation.

For this purpose, you will need to maintain a *list* of messages in the database.

About lists
In Opa, lists are one of the most common data structures. They are immutable linked lists.
Lists have type `list`. More precisely, a list of elements of type `t` has type `list(t)`, pronounced "list of `t`". The empty list is written

```
[]
```

or, equivalently, `{nil}`. It has type `list('a)`, which means that it could be a list of anything. A list containing elements `x`, `y`, `z` is written

```
[x,y,z]
```

or, equivalently,

```
{hd = x; tl =
{hd = y; tl =
{hd = z; tl =
{nil}}}}
```

More generally, the definition of `list` in Opa is:

```
type list('a) = {nil}
              / {hd: 'a; tl: list('a)}
```

If you have a list `l` and you wish to construct a list starting with element `x` and continuing with `l`, you can write either

```
[x|l]
```

or, equivalently,

```
{hd = x; tl = l}
```

or, equivalently,

```
List.cons(x, l)
```

About loops
If you have a list `l` and wish to apply a function `f` to all elements if `l`, use function `List.iter`. This is one of the many loop functions of Opa.
Yes, in Opa, loops are just regular functions.

For bonus points, make sure that the log is displayed on a slightly different background color.

4.8.6 Multi-room chat

Now that you know how to manipulate `Server.simple_dispatch`, you can implement a multi-room chat, with the following definition:

- visiting a page with path p connects you to a chatroom p;
- each message also contains the name of the chatroom;
- for a client visiting path p, only display messages for chatroom p.

Minimizing communications
The best place for deciding whether to display a message is inside the callback added with `Network.add_callback`. Fine-tuning this callback can help you minimize the amount of server-to-client communications. For this purpose, you can use directive `@server`, a variant of `@publish` which lets you force a function to be executed only on the server.

Scaling up
For optimal scalability, a better design is required.
You will need to maintain a family of networks, one for each room, typically as a `stringmap` of networks. Networks are real-time data structures and can therefore not be stored in the database, as it is meaningless to store information that becomes inconsistent and unsafe whenever a client disconnects. Rather, the `stringmap` should be maintained as part of the *state* of a *distributed session*. We will introduce this mechanism of *distributed sessions*, which is the powerful primitive used to implement networks, in a few chapters.

4.8.7 And more

Improve the wiki and the chat. Add features, make them nicer, make them better! And, once again, do not forget to show your changes to the community!

Chapter 5

Hello, web services

Nowadays, numerous web applications offer their features as *web services*, APIs that can be used by other web applications or native clients. This is how Twitter, GitHub, Google Maps or countless others can be scripted by third-party applications, using cleanly defined and easily accessible protocols.

With Opa, offering a web service is just as simple as creating any other form of web application. In this chapter, instead of writing a new application, we will extend our wiki to make it accessible through such a web API. This task will lead us through REST web service design, command-line testing of Opa services, management of URI queries and more.

Tip
Several protocols share the landscape of web services, in particular *REST* (Representational State Transfer, a simple standard which does not specify how messages should be formated, only how they should be exchanged), *SOAP* (a more complex standard imposing conventions on the formatting of messages) and *WSDL* (a higher-level protocol). In this chapter, we'll only cover *REST*.

5.1 Overview

In this chapter, we will modify our wiki to make it accessible by a *REST* API. This involves few changes from the original wiki, only the addition of a few cases to differentiate between several kinds of requests that can be sent by a client — which does not need to be a browser anymore.

If you are curious, this is the full source code of the REST wiki server:

```
import stdlib.web.template

db /wiki: stringmap(Template.default_content)
db /wiki[_] = Template.text("This page is empty"): Template.default_content

@publish load_source(topic)   = Template.to_source(Template.default, /wiki[topic])
@publish load_rendered(topic) = Template.to_xhtml( Template.default, /wiki[topic])
@publish save_source(topic, source) =
   match Template.try_parse(Template.default, source) with
    | ~{success}    -> do /wiki[topic] <- success; Template.to_xhtml(Template.default,  ↩
       success)
    | {failure = _} -> do /wiki[topic] <- Template.text(source); <>Error: {source}</>
remove_topic(topic) = Db.remove(@/wiki[topic])

edit(topic) =
   do Dom.set_value(#edit_content, load_source(topic))
   do Dom.hide(#show_content)
   do Dom.show(#edit_content)
   do Dom.give_focus(#edit_content)
```

```
   void

save(topic) =
   content = save_source(topic, Dom.get_value(#edit_content))
   do Dom.transform([#show_content <- content]);
   do Dom.hide(#edit_content);
   do Dom.show(#show_content);
   void

display(topic) =
   Resource.styled_page("About {topic}", ["/resources/css.css"],
     <div id=#header><div id=#logo></div>About {topic}</div>
     <div class="show_content" id=#show_content ondblclick={_ -> edit(topic)}>
          {load_rendered(topic)}
     </>
     <textarea class="edit_content" id=#edit_content style="display:none"
          cols="40" rows="30" onblur={_ -> save(topic)}></>
   )

rest(topic) =
(
  match HttpRequest.get_method() with
    | {some = method} ->
       match method with
         | {post}   -> _ = save_source(topic, HttpRequest.get_body()?"") Resource. ↩
            raw_status({success})
         | {delete} -> do remove_topic(topic) Resource.raw_status({success})
         | {get}    -> Resource.raw_response(load_source(topic), "text/plain", {success})
         | _ -> Resource.raw_status({method_not_allowed})
       end
    | _ -> Resource.raw_status({bad_request})
)

topic_of_path(path) = String.capitalize(String.to_lower(List.to_string_using("", "", "::",  ↩
    path)))

start =
   | {path = [] ... }               -> display("Hello")
   | {path = ["_rest_" | path] ...} -> rest(topic_of_path(path))
   | {~path ...}                    -> display(topic_of_path(path))

server = Server.of_bundle([@static_include_directory("resources")])
server = Server.simple_dispatch(start)
```

Run

We will now walk through the concepts introduced in this listing.

5.2 Removing topics

In the rest of this chapter (pun intended), we will want to be able to delete a topic previously added to the wiki. Adding this feature (without showing it in the user interface) is just the matter of one line, as follows:

```
remove_topic(topic) = Db.remove(@/wiki[topic])
```

In this extract, we use function `Db.remove`, a function whose sole role is to remove the contents of a database path. Notice the `@` before `/wiki[topic]`? This symbol signifies that we are not working with the *value* `/wiki[topic]` but with the *path* itself. If we had omitted this symbol, the Opa compiler would have complained that `Db.remove` cannot work with a `string` — which is absolutely true.

5.3 Resting a little

A web service behaves much like a web application, without the client part. In other word, as any Opa web application, it starts with a `server`:

```
topic_of_path(path) = String.capitalize(String.to_lower(List.to_string_using("", "", "::",  ↩
    path)))

start =
   | {path = [] ... }               -> display("Hello")
   | {path = ["_rest_" | path] ...} -> rest(topic_of_path(path))
   | {~path ...}                    -> display(topic_of_path(path))

server = Server.of_bundle([@static_include_directory("resources")])
server = Server.simple_dispatch(start)
```

In this version of `start`, we have slightly altered our pattern-matching to handle the case of paths starting with `"_rest_"`. We decide that such paths are actually entry points for REST-based requests and handle them as such. Here, we delegate the management to function `rest`, which we write immediately:

As you may see, this function is also quite simple:

```
rest(topic) =
(
  match HttpServer.get_method() with
    | {some = {post}} ->
         _ = save_source(topic,
           match HttpServer.get_body() with
             | ~{some} -> some
             |  {none} -> ""
         )
         Resource.raw_status({success})
    | {some = {delete}}->
         do remove_topic(topic)
         Resource.raw_status({success})
    | {some = {get}}   ->
         Resource.raw_response(load_source(topic), "text/plain", {success})
    | _ ->
         Resource.raw_status({method_not_allowed})
)
```

Warning
(API is currently evolving, update to new API)

First, notice that `rest` is based on pattern-matching. Expect to meet pattern-matching constantly in Opa. The first three patterns are built from some of to the distinct verbs of the standard vocabulary of REST (these verbs are called *Http methods*):

- `{post}` is used to place information on a server, here to add some content to the wiki;
- `{delete}` is used to remove information from the server, here remove a topic from the wiki;
- `{get}` is used to get information from a server, here to download the source code of an entry.

From these verbs, we build the following patterns:

- `{some = {post}}`, i.e. the Http method is defined and is *post*;

- `{some = {delete}}`, i.e. the Http method is defined and is *delete*;
- `{some = {get}}`, i.e. the Http method is defined and is *get*;
- _, i.e. any other case, whether the Http method is not defined or whether it is a method that we do not wish to handle.

Everything else in `rest` is simply function calls. You can find the definition of each function in the API documentation, so we will just introduce quickly the functions you have not seen yet:

- Function `HttpRequest.get_method` has type `-> option(method)`. If the function is called from a request and this request has a method *m*, it produces `{some = m}`. Otherwise, it produces `{none}`.
- Similarly, `HttpRequest.get_body` has type `-> option(string)`. If the function is called from a request containing a body *b*, it produces `{some = b}`. Otherwise, it produces `{none}`.
- Function `Resource.raw_response` has type `string, string, status -> resource`. It produces a resource with a body from its body, its MIME type, and a status. This function is commonly used to reply to REST requests.
- Finally, function `Resource.raw_status` has type `status -> resource`. It produces an empty resource, and is generally used to return an error to a REST request.

As pattern-matching against an `option` is very common, Opa provides an operator `?` that can be used to make the above extract shorter and more readable. Expression `a?b` is equivalent to the following three lines:

```
match a with
  | {none}  -> b
  | ~{some} -> some
```

With this expression, we may rewrite our extract as follows:

```
rest(topic) =
(
  match HttpServer.get_method() with
    | {some = {post}} ->
        _ = save_source(topic, HttpServer.get_body()?"")
        Resource.raw_status({success})
    | {some = {delete}}->
        do remove_topic(topic)
        Resource.raw_status({success})
    | {some = {get}}   ->
        Resource.raw_response(load_source(topic), "text/plain", {success})
    | _ ->
        Resource.raw_status({method_not_allowed})
)
```

And with this, we are done! Our wiki can now be scripted by external web applications:

```
import stdlib.web.template

db /wiki: stringmap(Template.default_content)
db /wiki[_] = Template.text("This page is empty"): Template.default_content

@publish load_source(topic)   = Template.to_source(Template.default, /wiki[topic])
@publish load_rendered(topic) = Template.to_xhtml( Template.default, /wiki[topic])
@publish save_source(topic, source) =
   match Template.try_parse(Template.default, source) with
    | ~{success}    -> do /wiki[topic] <- success; Template.to_xhtml(Template.default,  ↩
       success)
    | {failure = _} -> do /wiki[topic] <- Template.text(source); <>Error: {source}</>
remove_topic(topic) = Db.remove(@/wiki[topic])

edit(topic) =
```

```
   do Dom.set_value(#edit_content, load_source(topic))
   do Dom.hide(#show_content)
   do Dom.show(#edit_content)
   do Dom.give_focus(#edit_content)
   void

save(topic) =
   content = save_source(topic, Dom.get_value(#edit_content))
   do Dom.transform([#show_content <- content]);
   do Dom.hide(#edit_content);
   do Dom.show(#show_content);
   void

display(topic) =
   Resource.styled_page("About {topic}", ["/resources/css.css"],
     <div id=#header><div id=#logo></div>About {topic}</div>
     <div class="show_content" id=#show_content ondblclick={_ -> edit(topic)}>
          {load_rendered(topic)}
     </>
     <textarea class="edit_content" id=#edit_content style="display:none"
          cols="40" rows="30" onblur={_ -> save(topic)}></>
   )

rest(topic) =
(
  match HttpRequest.get_method() with
    | {some = method} ->
       match method with
         | {post}   -> _ = save_source(topic, HttpRequest.get_body()?"") Resource. ↩
             raw_status({success})
         | {delete} -> do remove_topic(topic) Resource.raw_status({success})
         | {get}    -> Resource.raw_response(load_source(topic), "text/plain", {success})
         | _ -> Resource.raw_status({method_not_allowed})
       end
    | _ -> Resource.raw_status({bad_request})
)

topic_of_path(path) = String.capitalize(String.to_lower(List.to_string_using("", "", "::",  ↩
    path)))

start =
   | {path = [] ... }               -> display("Hello")
   | {path = ["_rest_" | path] ...} -> rest(topic_of_path(path))
   | {~path ...}                    -> display(topic_of_path(path))

server = Server.of_bundle([@static_include_directory("resources")])
server = Server.simple_dispatch(start)
```

Run

All in all, the changes required a dozen lines of code.

Exercises will show you how to introduce more complex forms of scripting.

5.4 Testing it

The simplest way of testing a REST API is to use a command-line tool that lets you place requests directly, for instance `curl` or `wget`. Assuming that `curl` is installed on your system, the following command-line will test the result of placing a `{get}` request at address `_rest_/hello`:

```
curl localhost:8080/_rest_/hello
```

Execute this command-line and `curl` will show you the result of the call.

Similarly, the following command-line will test the result of placing a `{post}` request at the same address:

```
curl localhost:8080/_rest_/hello -d "I've just POSTed to change the contents of my wiki"
```

Now, we are not here to learn about `curl`, but to learn about Opa. And what best way to test the REST API of a wiki than by writing a web front-end that does not rely on its own database but on that of the wiki we have just defined?

We will do just this in the next chapter.

5.5 Questions

5.5.1 When is a method or a body not defined?

As mentioned, functions `HttpServer.get_method` and `HttpServer.get_body` can produce result `{none}` if the http method (respectively the body) does not exist.

This may be surprising, as, by definition of the protocols, every request has a method (not all have a body). Indeed, the only case in which `HttpServer.get_method` returns `{none}` is when there is *no request*, i.e. when the function has been called by the server for its own use and not during the execution of a request on behalf of a web browser or a distant web server.

On the other hand, many requests do not have a body. Function `HttpServer.get_body` returns `{none}` when there is no body, or when there is no request, as above.

5.5.2 Only one `server`?

If you have started thinking about large applications, at this stage, you might start worrying about having to centralize all your path management into only one pattern-matching, which could hurt modularity and hamper your work.

Well, you can stop worrying. With Opa, you may combine any number of servers in an application. If you look at the API documentation of Opa, you can find function `Server.make_dispatch`, which is designed to let you write separate servers for subpaths, subdomains, etc.

5.6 Exercises

5.6.1 Rest for chat

Add a REST API to your chat, with the following feature:

- use a `{post}` request send a message for immediate display into the chat (for the moment, we will assume that the message has been written by author "ghost").

Tip
To deal with several entry points, you will have to rewrite your `server` and replace `one_page_bundle` by a dispatcher. For these exercises, we decide that any request placed on path `_rest_` is a REST request.

For testing, use the following command-line (assuming that `curl` is installed on your system):

```
curl localhost:8080/_rest_ -d "Whispers..."
```

5.6.2 Rest for chat, with logs

If you have not done so yet, update your chat to maintain conversation logs in the database.

Now, add the following REST API:

- use a `{get}` request to get the log of messages as `string` containing one message per line.

Remember, use function `List.to_string_using` to convert a list to a `string`.

5.6.3 Rest for chat, with queries

For this exercise, we wish to extend the REST API for the chat to be able to send a message *and* give a name to the author of the message. For this purpose, we need to send more informations than simply `{post}`. In the REST world, there are typically two ways of passing additional informations: either in the URI itself or in the body of the request. For this exercise, we will see the first option:

- if a `{post}` request is received on `_rest_` and if the *query* of the request contains a pair `("author", x)`, use the value of `x` as the author name;
- otherwise, use name "ghost", as above.

About queries
A *query* is an element of a URI. From the user's perspective, queries look like `?author=name&arg2=val2&arg3=val3`. From the developer's perspective, the query is contained in field `query` of the URI, just as `path`. This field contains a list of pairs with the name of the argument and its value. So, for the previous query, the list will look like:

```
[("author", "name"), ("arg2", "val2"), ("arg3", "val3")]
```

Note that the order of these arguments is meaningless.

About association lists
Lists of pairs containing a name and a value (or, more generally, a key and a value) are generally called "association lists".
In Opa, the most common function to extract a value from an association list is `List.assoc`. This function takes two arguments: the key to search and the list in which to search. Its result is an `option` which may contain either `{none}` (if the key does not appear in the list) or `{some = v}` (if the key appears in the list, associated to value `v`).

5.6.4 Rest for chat, with JSON

Another common technique used among REST services is to pass additional information as part of the body of the request, often formated using the JavaScript Object Notation language (or *JSON*). The objective of this exercise is to use JSON instead of the URI to send the author name to the server.

- if a `{post}` request is received on `_rest_` and if the body of the request is a valid JSON construction containing a field `"author"`, use the value associated to this field as the author name;
- otherwise, use name "ghost", as above.

JSON requests
To obtain the JSON body of a request, use function `HttpRequest.get_json_body`.

About JSON
JSON is a format of strings which can be interpreted as simple data structures. In Opa, a string in JSON format can be transformed into a value with type `RPC.Json.json` by using function

```
Json.deserialize: string -> option(RPC.Json.json)
```

Note that this function can return `{none}` if the `string` was incorrectly formated.
The opposite operation is implemented by function

```
Json.serialize: RPC.Json.json -> string
```

Type `RPC.Json.json` is defined as follows:

```
type RPC.Json.json =
    { Int:    int}
  / { Float:  float}
  / { String: string}
  / { Bool:   bool}
  / { List:   list(RPC.Json.json)}
  / { Record: list((string, RPC.Json.json))}
```

As above, case `Record` corresponds to a list of associations.

Chapter 6

Hello, web services client

With Opa, accessing a distant web service is as simple as creating one. In this chapter, we will develop a variant of our wiki which, instead of using its own database, will serve as a front-end for the wiki developed in the previous chapter. This task will lead us through the other side of REST: how to connect to a distant server, send commands and interpret results. Somewhere along the way, we will also see how to handle command-line arguments in Opa, how to analyze text and some interesting features of the language.

6.1 Overview

The general idea behind REST is to use the well-known HTTP protocol to send/receive commands through the web. In other words, a REST client is just a web application that has a few of the features of a browser, i.e. a web client: the functions that we will meet in this chapter can be used just as well for purposes unrelated to REST, for instance to write a web crawler, to post the contents of a web form automatically, or to download a distant image from an Opa application.

In this chapter we modify further our wiki to make it use a distant *REST* API instead of its own database. As previously, this involves few changes from the original wiki: we remove the database, we handle error cases in case of communication issues, and we introduce command-line options to let users specify where to find the distant REST server.

If you are curious, this is the full source code of the REST wiki client (which also acts as a server):

```
import stdlib.web.template

uri_for_topic =
    domain_parser = {CommandLine.default_parser with
          names       = ["--wiki-server-domain"]
          description = "The REST server for this wiki. By default, localhost."
          on_param(x)     = parser y=Rule.consume -> {no_params = {x with domain = y}}
    }
    port_parser = {CommandLine.default_parser with
         names       = ["--wiki-server-port"]
         description = "The server port of the REST server for this wiki. By default,  ↩
             8080."
         on_param(x)     = parser y=Rule.natural -> {no_params = {x with port = {some = y ↩
             }}}
    }
    base_uri = CommandLine.filter({title     = "Wiki arguments"
     init      = {Uri.default_absolute with domain = "localhost"
                                            schema = {some = "http"}}
     parsers   = [domain_parser, port_parser]
     anonymous = []
    })
    topic -> Uri.of_absolute({base_uri with path   = ["_rest_", topic]})

@publish load_source(topic)    =
```

```
   match WebClient.Get.try_get(uri_for_topic(topic)) with
      | {failure = _} -> "Error, could not connect"
      | {~success}    -> match WebClient.Result.get_class(success) with
          | {success} -> success.content
          | _         -> "Error {success.code}"
   end

@publish load_rendered(topic) =
    source = load_source(topic)
    match Template.try_parse( Template.default, source) with
      | {failure = _} -> <>{source}</>
      | ~{success}-> Template.to_xhtml(Template.default, success)

@publish save_source(topic, source) =
   match Template.try_parse(Template.default, source) with
    | ~{success}    -> match WebClient.Post.try_post(uri_for_topic(topic), source) with
         | { failure = _ } -> {failure = "Could not reach distant server"}
         | { success = s } -> match WebClient.Result.get_class(s) with
              |  {success} -> {success = Template.to_xhtml(Template.default, success)}
              |  _         -> {failure = "Error {s.code}"}
           end
        end
    | {failure = _} -> {failure = "Incorrect syntax"}

remove_topic(topic) =
   _ = WebClient.Delete.try_delete(uri_for_topic(topic))
   void

edit(topic) =
   do Dom.transform([#show_messages <- <></>])
   do Dom.set_value(#edit_content, load_source(topic))
   do Dom.hide(#show_content)
   do Dom.show(#edit_content)
   do Dom.give_focus(#edit_content)
   void

save(topic) =
   match save_source(topic, Dom.get_value(#edit_content)) with
     | { ~success } ->
       do Dom.transform([#show_content <- success]);
       do Dom.hide(#edit_content);
       do Dom.show(#show_content);
       void
     | {~failure} ->
       do Dom.transform([#show_messages <- <>{failure}</>])
       void

display(topic) =
   Resource.styled_page("About {topic}", ["/resources/css.css"],
     <div id=#header><div id=#logo></div>About {topic}</div>
     <div class="show_content" id=#show_content ondblclick={_ -> edit(topic)}>
        {load_rendered(topic)}
     </>
     <div class="show_messages" id=#show_messages />
     <textarea class="edit_content" id=#edit_content style="display:none"
        cols="40" rows="30" onblur={_ -> save(topic)}></>
   )

rest(topic) =
(
  match HttpRequest.get_method() with
    | {some = method} ->
```

```
       match method with
         | {post}   ->
             _ = save_source(topic, HttpRequest.get_body()?"")
             Resource.raw_status({success})
         | {delete} ->
             do remove_topic(topic)
             Resource.raw_status({success})
         | {get}    -> Resource.source(load_source(topic), "text/plain")
         | _ -> Resource.raw_status({method_not_allowed})
       end
    | _ -> Resource.raw_status({bad_request})
)

topic_of_path(path) = String.capitalize(String.to_lower(
                    List.to_string_using("", "", "::", path)
))

start =
   | {path = [] ... }              -> display("Hello")
   | {path = ["rest" | path] ...} -> rest(topic_of_path(path))
   | {~path ...}                   -> display(topic_of_path(path))

server = Server.of_bundle([@static_include_directory("resources")])
server = Server.simple_dispatch(start)
```

Run

6.2 The web client

To connect to distant servers and services, Opa offers a module called `WebClient`. The following extract adapts `load_source` to perform loading from a distant service:

```
@publish load_source(topic)   =
  match WebClient.Get.try_get(uri_for_topic(topic)) with
         | {failure = _} -> "Error, could not connect"
         | {~success}    -> match WebClient.Result.get_class(success) with
                | {success} -> success.content
                | _         -> "Error {success.code}"
  end
```

As in previous variants of the wiki, this version of `load_source` attempts to produce the source code matching a topic. The main difference is that, instead of reading the database, it performs a `{get}` request on a distant server. This is the role of function `WebClient.Get.try_get` — of course, module `WebClient` offers similar functions other operations other than `{get}`. This function takes as argument a URI — here, provided by a function `uri_for_topic` that we will need to write at some point — and produces as result a sum type, containing either `{failure = f}` or `{success = s}`.

Failures take place when the operation could not proceed at all, for instance because of network issues, or because the distant server is down. In such case, `f` contains more details about the exact error. Any other case means that the request was successful. Note that, depending on what you are trying to do, the result of the request could still be something that is no use to your application. For instance, the server may have returned some content along with a status of "404 Not Found", to indicate that this content is a default page and that it actually does not know what to do with your URI. It could be a "100 Continue", to indicate that you should now send more information before it can proceed. All these responses are *successes* at the level of `WebClient`, although many applications decide to treat them as failures.

Here, for our simple protocol, we use function `WebClient.Result.get_class` to perform a rough decoding of the server response and categorize it as a success (case `{success}`) or anything else (redirection, client error, server error, etc.) (case _). In case of success, we return the content of the response, e.g. `success.content`.

There's more to distribution than REST
Do not forget that this web client is a demonstration of REST. In Opa, REST is but one of the many ways of handling distribution. Indeed, as long as your application is written only in Opa, Opa can perform distribution automatically, using protocols that are largely more efficient for this purpose than REST.

Function `remove_topic` is even simpler (we ignore the result of the operation):

```
remove_topic(topic) =
   _ = WebClient.Delete.try_delete(uri_for_topic(topic))
   void
```

We can similarly adapt `load_rendered`, with a slight change to use the API we have previously published:

```
@publish load_rendered(topic) =
    source = load_source(topic)
    match Template.try_parse( Template.default, source) with
      | {failure = _} -> <>{source}</>
      | ~{success}    -> Template.to_xhtml(Template.default, success)
```

Finally, we can adapt `save_source`, as follows:

```
@publish save_source(topic, source) =
   match Template.try_parse(Template.default, source) with
    | ~{success}    -> match WebClient.Post.try_post(uri_for_topic(topic), source) with
         | { failure = _ } -> {failure = "Could not reach distant server"}
         | { success = s } -> match WebClient.Result.get_class(s) with
              |  {success} -> {success = Template.to_xhtml(Template.default, success)}
              |  _         -> {failure = "Error {s.code}"}
           end
         end
    | {failure = _} -> {failure = "Incorrect syntax"}
```

This version of `save_source` differs slightly from the original, not only because it uses a `{post}` request to send the information, but also because it returns something slightly more precise than the original, for better error reporting.

We take this opportunity to tweak our UI with a box meant to report such errors:

6.3 Improving error reporting

We add a `<div>` called `show_messages` to the HTML-like user interface, and we update it in `edit` and `save`, as follows:

```
display(topic) =
   Resource.styled_page("About {topic}", ["/resources/css.css"],
     <div id=#header><div id=#logo></div>About {topic}</div>
     <div class="show_content" id=#show_content ondblclick={_ -> edit(topic)}>
       {load_rendered(topic)}
     </>
     <div class="show_messages" id=#show_messages />
     <textarea class="edit_content" id=#edit_content style="display:none"
        cols="40" rows="30" onblur={_ -> save(topic)}></>
   )

edit(topic) =
   do Dom.transform([#show_messages <- <></>]) //Clear messages
   do Dom.set_value(#edit_content, load_source(topic))
   do Dom.hide(#show_content)
   do Dom.show(#edit_content)
   do Dom.give_focus(#edit_content)
```

```
   void

save(topic) =
   match save_source(topic, Dom.get_value(#edit_content)) with
     | { ~success } ->
       do Dom.transform([#show_content <- success]);
       do Dom.hide(#edit_content);
       do Dom.show(#show_content);
       void
     | {~failure} ->
       do Dom.transform([#show_messages <- <>{failure}</>])
       void
```

And that is all for the user interface.

6.4 Working with URIs

We have already been using URIs by performing pattern-matching on them inside dispatchers. It is now time to build new URIs for our function `uri_for_topic`.

About absolute URIs
Many languages consider that a URI is simply a `string`. In Opa, URIs come in several flavors. So far, we have been using *absolute uris*, as defined by the following type:

```
type Uri.absolute =
    { schema:      option(string)
    ; credentials: Uri.uri_credentials
    ; domain:      string
    ; port:        option(int)
    ; path:        list(string)
    ; query:       list((string,string))
    ; fragment:    option(string)
    }

type Uri.uri_credentials =
    { username : option(string)
    ; password : option(string)
    }
```

Other flavor exists, e.g. to handle e-mail addresses, relative URIs, etc.
The most general form of URI is `Uri.uri`, whose definition looks like:

```
type Uri.uri = Uri.absolute / Uri.relative / ...
```

To cast an `Uri.absolute` into a `Uri.uri`, use function `Uri.of_absolute`. To build a `Uri.absolute`, you can either construct a record manually or *derive* one from `Uri.default_absolute`.

To match the API we have defined earlier, we need to place requests for topic `mytopic` at URI `http://myserver/_rest_/mytop`
In other words, we may write:

```
uri_for_topic(topic) =
  Uri.of_absolute({ schema      = {some = "http"}
    credentials = {username = {none} password = {none}}
    domain      = "localhost"  //Assume server is launched locally
    port        = {some = 8080}//Assume server is launched on port 8080
    path        = ["_rest_", mytopic]
    query       = []
    fragment    = {none}
  })
```

It is, however, a tad clumsy to provide `query`, `fragment`, `port`, etc. only to mention that they are not used. So we will prefer to *derive* a uri from `Uri.default_absolute`, as follows:

```
uri_for_topic(topic) =
  Uri.of_absolute({Uri.default_absolute with
     schema = {some = "http"}
     domain = "localhost"
     port   = {some = 8080}
     path   = ["_rest_", topic]
  })
```

Record derivation
Use *record derivation* to construct a record from another one by modifying several fields. For instance, if we have

```
foo = {a = 1; b = 2}
```

we may write

```
bar = {foo with b = 17}
```

This is equivalent to the following

```
bar = {a = foo.a; b = 17}
```

Using record derivation is a good habit, as it is not only more readable than copying field values from one record to another, but also faster.

With this, your client wiki is complete:

```
import stdlib.web.template

db /wiki: stringmap(Template.default_content)
db /wiki[_] = Template.text("This page is empty"): Template.default_content

@publish load_source(topic)   = Template.to_source(Template.default, /wiki[topic])
@publish load_rendered(topic) = Template.to_xhtml( Template.default, /wiki[topic])
@publish save_source(topic, source) =
   match Template.try_parse(Template.default, source) with
    | ~{success}    -> do /wiki[topic] <- success; Template.to_xhtml(Template.default,  ↩
       success)
    | {failure = _} -> do /wiki[topic] <- Template.text(source); <>Error: {source}</>
remove_topic(topic) = Db.remove(@/wiki[topic])

edit(topic) =
   do Dom.set_value(#edit_content, load_source(topic))
   do Dom.hide(#show_content)
   do Dom.show(#edit_content)
   do Dom.give_focus(#edit_content)
   void

save(topic) =
   content = save_source(topic, Dom.get_value(#edit_content))
   do Dom.transform([#show_content <- content]);
   do Dom.hide(#edit_content);
   do Dom.show(#show_content);
   void

display(topic) =
   Resource.styled_page("About {topic}", ["/resources/css.css"],
     <div id=#header><div id=#logo></div>About {topic}</div>
     <div class="show_content" id=#show_content ondblclick={_ -> edit(topic)}>
```

```
          {load_rendered(topic)}
     </>
     <textarea class="edit_content" id=#edit_content style="display:none"
          cols="40" rows="30" onblur={_ -> save(topic)}></>
   )

rest(topic) =
(
  match HttpRequest.get_method() with
    | {some = method} ->
       match method with
         | {post}   -> _ = save_source(topic, HttpRequest.get_body()?"") Resource. ↩
            raw_status({success})
         | {delete} -> do remove_topic(topic) Resource.raw_status({success})
         | {get}    -> Resource.raw_response(load_source(topic), "text/plain", {success})
         | _ -> Resource.raw_status({method_not_allowed})
       end
    | _ -> Resource.raw_status({bad_request})
)

topic_of_path(path) = String.capitalize(String.to_lower(List.to_string_using("", "", "::",  ↩
    path)))

start =
   | {path = [] ... }               -> display("Hello")
   | {path = ["_rest_" | path] ...} -> rest(topic_of_path(path))
   | {~path ...}                    -> display(topic_of_path(path))

server = Server.of_bundle([@static_include_directory("resources")])
server = Server.simple_dispatch(start)
```

Run

Launch the server wiki, launch the client wiki on a different port (use option `-p` or `--opa-server-port` to select a port) and behold, you can edit your wiki from two distinct ports. Or two distinct servers, if you replace `"localhost"` by the appropriate server name.

On the other hand, replacing a magic constant by another equally magic constant is not very nice. Would it not be better to decide that the server name and port are options that can be configured without recompiling?

6.5 Handling options

Opa is a higher-order language. Among other things, this means that there are many ways of defining a function. So far, our function definitions have been quite simple, but if we wish to define a function whose behavior depends on a command-line option or on an option somehow defined at start-up, the best and nicest way is to expand our horizon.

In this case, expanding our horizon starts by rewriting `uri_for_topic` as follows:

```
uri_for_topic = topic ->
(
 Uri.of_absolute({Uri.default_absolute with
     schema = {some = "http"}
     domain = "localhost"
     port   = {some = 8080}
     path   = ["_rest_", topic]
  })
)
```

So far, this is absolutely equivalent to what we had written earlier. Note that we can move the parenthesis as follows:

```
uri_for_topic = (topic ->
 Uri.of_absolute({Uri.default_absolute with
     schema = {some = "http"}
     domain = "localhost"
     port   = {some = 8080}
     path   = ["_rest_", topic]
  })
)
```

While we have not changed the behavior of the function at all, this rewrite is a nice opportunity to split the construction URI in two parts, as follows:

```
uri_for_topic =
(
  base_uri = {Uri.default_absolute with
     schema = {some = "http"}
     domain = "localhost"
     port   = {some = 8080}
  }
  topic -> Uri.of_absolute({base_uri with
     path   = ["_rest_", topic]
  })
)
```

Suddenly, things have changed a little: `uri_for_topic` is still a function that takes a `topic` and returns a URI, but with a twist. At some point, when the function itself is built, it first initializes a (local) value called `base_uri` which it uses whenever the function is called. This is an example use of *closures*.

About closures

You have already met closures in previous chapters. Indeed, most of the event handlers we have been using so far are closures. Rigorously, a *closure* is a function which uses some values that are local but defined outside of the function itself. Closures are a very powerful mechanism used in many places in Opa, in particular for event handlers.

With this rewrite, the only task we still have ahead of us is changing `base_uri` so that it uses options specified on the command-line or in an option file. For both purposes, Opa offers a module `CommandLine`:

```
uri_for_topic =
(
  default_uri = {Uri.default_absolute with domain = "localhost"
                                          schema = {some = "http"}}
  base_uri = CommandLine.filter({
     title     = "Wiki arguments"
     init      = default_uri
     parsers   = []
     anonymous = []
  })
  topic -> Uri.of_absolute({base_uri with
     path   = ["_rest_", topic]
  })
)
```

This variant on `uri_for_topic` calls `CommandLine.filter` to instruct the option system to take into account a family of arguments to progressively construct `base_uri`, starting from `default_uri`. We name this family "*Wiki arguments*" and we specify its behavior with fields `parsers` (used for named arguments) and `anonymous` (used for anonymous arguments) which are both empty for the moment. As long as both fields are empty, this family has no effect and `base_uri` is always going to be equal to `default_uri` — we will change this shortly. Also, for the moment, if you compile your application and launch it with command-line argument `--help`, you will see an empty entry for a family called "Wiki arguments".

Let us add one command-line option (or, more precisely, a *command-line parser*) to our family, as follows:

```
  option_port = {CommandLine.default_parser with
     names           = ["--wiki-server-port"]
     on_param(x)     = parser y=Rule.natural -> {no_params = {x with port = {some = y}}}
  }
```

As you can see, a command-line parser is a record (it has type `CommandLine.parser`), and here, we derive it from `CommandLine.d`
In this extract, we only specify the bare minimum.

Firstly, a command-line parser should have at least one name, here "--wiki-server-port".

Secondly, Opa needs to know what it should do whenever it encounters something along the lines of "--wiki-server-port foo" on the command-line. This is the role of field `on_param`. Argument `x` is the value we are currently building — here, initially, `default_uri`. The body of this field is a *text parser*, i.e. a construction that should analyze a text and either extract information or reject it. Here, we just want a non-negative integer (aka a "natural number"), a construction for which the library offers a predefined text parser called `Rule.natural`. We call the result `y`.

About text parsers
Opa offers a powerful text analysis feature with text parsers. Text parsers have roughly the same role as regular expressions engines found in many web-related languages, but they are considerably more powerful.
A text parser is introduced with keyword `parser` and has a syntax roughly comparable to pattern-matching:

```
 parser y=Rule.natural -> //do something with y
      | y=Rule.hex     -> //do something with y
      | "none"         -> //...
```

This parser will accept any non-negative integer and execute the first branch, or any hexadecimal integer and execute the second branch, or the character string `"none"` and execute the third branch. If none of the branches matches the text, parsing fails.
The core function for applying a text parser to some text is `Parser.try_parse`. You can find a number of predefined parsing functions in module `Rule`. Additional modules offer custom parsing, e.g. `Uri.uri_parser`.

The result of `on_param` must have one of three shapes:

- `{no_params = v}`, if the option parser does not expect any additional argument and is now ready to produce value `v`;
- `{params = v}`, if the option parser expects at least one other argument;
- `{opt_params = v}`, if the option parser can handle additional arguments but is also satisfied if no such argument is provided.

Here, we expect only one argument after "--wiki-server-port" so we just produce a value with `{no_params = ...}`. As for the result itself, we derive from `x` the same absolute URI, but with a new content in field `port`.

We can now define in the exact same manner the command-line parser for the host:

```
   host_parser = {CommandLine.default_parser with
         names       = ["--wiki-server-host"]
         on_param(x) = parser y=Rule.consume -> {no_params = {x with domain = y}}
   }
```

The main difference is that we use predefined text parser `Rule.consume` (which accepts anything) instead of `Rule.natural` (which only accepts non-negative integers).

Once we have added both our parsers to `parsers`, we are ready. With a little additional documentation, we obtain:

```
uri_for_topic =
    host_parser = {CommandLine.default_parser with
          names       = ["--wiki-server-host"]
          description = "The host of a the server offering a REST API for this wiki. By ↩
              default, localhost."
```

```
          on_param(x) = parser y=Rule.consume -> {no_params = {x with domain = y}}
    }
    port_parser = {CommandLine.default_parser with
         names       = ["--wiki-server-port"]
         description = "The port of a the server offering a REST API for this wiki. By  ↩
            default, 8080."
         on_param(x) = parser y=Rule.natural -> {no_params = {x with port = {some = y}}}
    }
    base_uri = CommandLine.filter({
     title     = "Wiki arguments"
     init      = {Uri.default_absolute with domain = "localhost"
                                            schema = {some = "http"}}
     parsers   = [domain_parser, port_parser]
     anonymous = []
    })
    topic -> Uri.of_absolute({base_uri with path   = ["_rest_", topic]})
```

This completes our REST client. We now have a full-featured REST client that can also act as a server and supports command-line configuration.

The full source code follows:

```
import stdlib.web.template

uri_for_topic =
    domain_parser = {CommandLine.default_parser with
          names       = ["--wiki-server-domain"]
          description = "The REST server for this wiki. By default, localhost."
          on_param(x)     = parser y=Rule.consume -> {no_params = {x with domain = y}}
    }
    port_parser = {CommandLine.default_parser with
         names       = ["--wiki-server-port"]
         description = "The server port of the REST server for this wiki. By default,  ↩
            8080."
         on_param(x)     = parser y=Rule.natural -> {no_params = {x with port = {some = y ↩
            }}}
    }
    base_uri = CommandLine.filter({title     = "Wiki arguments"
     init      = {Uri.default_absolute with domain = "localhost"
                                            schema = {some = "http"}}
     parsers   = [domain_parser, port_parser]
     anonymous = []
    })
    topic -> Uri.of_absolute({base_uri with path   = ["_rest_", topic]})

@publish load_source(topic)   =
   match WebClient.Get.try_get(uri_for_topic(topic)) with
      | {failure = _} -> "Error, could not connect"
      | {~success}    -> match WebClient.Result.get_class(success) with
          | {success} -> success.content
          | _         -> "Error {success.code}"
   end

@publish load_rendered(topic) =
    source = load_source(topic)
    match Template.try_parse( Template.default, source) with
      | {failure = _} -> <>{source}</>
      | ~{success}-> Template.to_xhtml(Template.default, success)

@publish save_source(topic, source) =
   match Template.try_parse(Template.default, source) with
    | ~{success}    -> match WebClient.Post.try_post(uri_for_topic(topic), source) with
```

```
         | { failure = _ } -> {failure = "Could not reach distant server"}
         | { success = s } -> match WebClient.Result.get_class(s) with
              |  {success} -> {success = Template.to_xhtml(Template.default, success)}
              |  _         -> {failure = "Error {s.code}"}
           end
        end
    | {failure = _} -> {failure = "Incorrect syntax"}

remove_topic(topic) =
   _ = WebClient.Delete.try_delete(uri_for_topic(topic))
   void

edit(topic) =
   do Dom.transform([#show_messages <- <></>])
   do Dom.set_value(#edit_content, load_source(topic))
   do Dom.hide(#show_content)
   do Dom.show(#edit_content)
   do Dom.give_focus(#edit_content)
   void

save(topic) =
   match save_source(topic, Dom.get_value(#edit_content)) with
     | { ~success } ->
       do Dom.transform([#show_content <- success]);
       do Dom.hide(#edit_content);
       do Dom.show(#show_content);
       void
     | {~failure} ->
       do Dom.transform([#show_messages <- <>{failure}</>])
       void

display(topic) =
   Resource.styled_page("About {topic}", ["/resources/css.css"],
     <div id=#header><div id=#logo></div>About {topic}</div>
     <div class="show_content" id=#show_content ondblclick={_ -> edit(topic)}>
        {load_rendered(topic)}
     </>
     <div class="show_messages" id=#show_messages />
     <textarea class="edit_content" id=#edit_content style="display:none"
        cols="40" rows="30" onblur={_ -> save(topic)}></>
   )

rest(topic) =
(
  match HttpRequest.get_method() with
    | {some = method} ->
       match method with
         | {post}   ->
             _ = save_source(topic, HttpRequest.get_body()?"")
             Resource.raw_status({success})
         | {delete} ->
             do remove_topic(topic)
             Resource.raw_status({success})
         | {get}    -> Resource.source(load_source(topic), "text/plain")
         | _ -> Resource.raw_status({method_not_allowed})
       end
    | _ -> Resource.raw_status({bad_request})
)

topic_of_path(path) = String.capitalize(String.to_lower(
                    List.to_string_using("", "", "::", path)
))
```

```
start =
   | {path = [] ... }              -> display("Hello")
   | {path = ["rest" | path] ...} -> rest(topic_of_path(path))
   | {~path ...}                  -> display(topic_of_path(path))

server = Server.of_bundle([@static_include_directory("resources")])
server = Server.simple_dispatch(start)
```

Run

6.6 Exercises

6.6.1 Database vs. REST

Modify the wiki so that it acts both as a database-backed wiki and as a REST client:

- by default, behave as the REST client wiki;
- whenever information is downloaded from the REST server, store the information to the local database;
- whenever information is updated locally, store the information to the local database and upload it to the REST server;
- if connection fails for some reason, fallback to the database.

6.6.2 Database vs. REST vs. command-line

Modify the wiki of the previous exercise so that:

- the REST server can be specified from the command-line;
- if no server is specified from the command-line, it behaves exactly as the non-REST wiki;
- otherwise, behave as the wiki of the previous exercise.

Using tuples
For this exercise, you may need to define not just one function using the command-line but several. In this case, it will probably be interesting to use a *tuple* definition, such as

```
(a, b) =
  x = 50
  (x, x+1)
```

This tuple definition defines both `a = 50` and `b = 51`. You can, of course, use more complex expressions instead of `50`.

6.6.3 Architecting a REST chat

How would you design a chat distributed among servers using only REST for communications between servers?

A REST chat?
While it is definitely possible to write a REST-based chat in Opa, this is not the preferred way of implementing a multi-server application. But it is an interesting exercise, if only to experience the contrast between manual REST-style distribution and automated Opa-style distribution.

Chapter 7

Hello, scalability

From the ground up, Opa was designed for scalability. This means that any application written in Opa can (almost) automatically take advantage of additional cores or additional servers to distribute treatment, storage or delivery. In this chapter, we will see how to adapt, deploy and execute our chat and our wiki in a distributed, load-balanced setting. As you will see, it is very, very simple.

7.1 Prerequisites

You are about to distribute instances of your server on other computers. To do so, you will need a valid account and an ssh connection. Firstly, make sure that you can connect through ssh to computers you want instances to run, without prompting for a password. Secondly, check whether these computers have **base64** installed. Thirdly, check whether **HAProxy** is installed on your localhost.

If you don't know how to do any of these steps, don't panic and have a look at the FAQ

It's now time to distribute !

7.2 Distributing Hello, chat

Done! We have a distributed Hello, chat.

Don't be confused, this is no mistake. All the versions of the chat we have been implementing so far are distributed. If you want to try any of them, you just need to launch it in distributed mode, with the following command-line:

```
opa-cloud --host localhost,2 hello_chat.exe
```

You can also explicit each host you want an instance to run on e.g. twice on *localhost*:

```
opa-cloud --host localhost --host localhost hello_chat.exe
```

Both lines are equivalent.

You can now connect to http://localhost:8080/ and users connecting to the service will be automatically distributed between two processes running on your computer. They can, of course, chat together, regardless of the server to which they are effectively connected. You are of course not limited to two processes or to a single computer: if you have other computers at hand (say, a cloud), with the same binary configuration, you can add distant hosts, too:

```
opa-cloud --host localhost,2 --host jeff@albertson hello_chat.exe
```

Not bad for, well, zero lines of code to add or modify!

7.3 Distributing Hello, wiki

We are not going to crack the same joke twice, but we could, though. Again, don't modify what you have already written for Hello, wiki and merely invoke the following command-line:

```
opa-cloud --host localhost,2 hello_wiki.exe
```

Users are dispatched to servers so as to balance the load between these servers. Of course, they share the same database. And, if you have modified the wiki so as to show updates as they take place, this feature will keep working in a distributed mode.

7.4 Examples:

The following command line distributes 6 instances of Hello, chat on `albertson`, with user `jeff`. Each instance will be listening to a port between 7000 et 7005 included.

```
opa-cloud --host jeff@albertson:7000,6 hello_chat.exe
```

The following command line distributes 10 instances of Hello, wiki on an Amazon EC2 instance with key `jeff.pem`. Each instance will listen to a port between 9090 and 9099. The HAProxy binary will be `~/bin/haproxy`.

```
opa-cloud hello_wiki.exe --host-pem ubuntu@ec-XXX.amazonaws.com:9090,10 --pem jeff.pem \
                         --haproxy ~/bin/haproxy
```

7.5 Frequently Asked Questions

7.5.1 How can Hello, chat work be distributed?

We have spent years making sure that it appears as magic, you don't want to ruin the magic now, do you?

For a few more details, we need to start with the network defined in Hello, Chat:

```
room = Network.cloud("room"): Network.network(message)
```

Function `Network.cloud` not only constructs a network, but also declares it to Opa's built-in distribution directory. In this directory, the network is called `"room"` (we could, of course, have given it any other name — some developers prefer writing `@pos`, which gives as name the position in the source code). Subsequent calls to `Network.cloud` by *any of the servers you just launched* will return the *same* network. Consequently, whenever a client calls `Network.add_callback` with `room`, the callback is known to all servers.

We will detail this further in the reference chapters.

Non-cloud networks
You can of course declare networks that you do not want to publish in the distribution directory. For this purpose, use function `Network.empty` rather than `Network.cloud`.

7.5.2 How does Hello, wiki work?

In a few words: it works because Opa's built-in database engine is fully distributed and compatible with `opa-cloud`.

7.5.3 How can check my ssh connection?

Let say your remote computer is called *albertson* and your user name is *jeff*. You should be able to connect to albertson with the following command line:

```
$ ssh jeff@albertson
jeff@albertson's password:
jeff@albertson ~ $
```

Let's get rid of the password prompt by adding your public key to the list of authorized keys. To do so, copy the content of /.ssh/id_dsa.pub to /.ssh/authorized_keys on *albertson*.

You can now log off *albertson*:

```
jeff@albertson ~ $ exit
logout
Connection to albertson closed.
$
```

7.5.4 How can I get rid of the prompt for password?

You must append your public key to the list of authorized keys of the remote host i.e. usually `~user/.ssh/authorized_keys`.

7.5.5 How can I make sure that my remote computer albertson provide base64 ?

To check whether *albertson* has this tool, simply type:

```
jeff@albertson ~ $ base64 --version
base64 1.5
Last revised: 10th June 2007
The latest version is always available
at http://www.fourmilab.ch/webtools/base64
```

Don't worry if you don't have the same version, the important thing is not to get anything like the following:

```
jeff@albertson ~ $ base64 --version
-bash: base64: command not found
```

But if you do, please see Getting Opa chapter.

7.5.6 How can I use this on a machine with a specific public-key?

You can ask `opa-cloud` to connect to servers with a specific public key with options *--pem* and *--host-pem*. The former specifies the key and the latter specifies a host which needs this key. Although you can define only one key, you can specify several hosts:

```
opa-cloud --pem ~/.ssh/mykey.pem --host localhost --host-pem user@sver hello_chat.exe
```

This command line will start two instances of hello_chat.exe, one on your localhost, and one on `sver` connecting to it with `mykey.pem`.

7.5.7 How can I use this with Amazon Web Services?

Opa's built-in distribution works very nicely with Amazon EC2 — and just as well with other public or private clouds. Amazon allows you to create a pair of key. You will use the public key to connect to your EC2 instance.

- start an instance and make sure to open ports 22, 8081 and 1086 (see the group security option in your AWS Management console)
- retrieve the url which should look like: ec2-XXX.amazonaws.com
- use `opa-cloud` to distribute your service on this instance:

```
opa-cloud --pem mykey.pem --host-pem ubuntu@ec2-XXX.amazonaws.com hello_chat.exe
```

You can of course distribute your service on more then one instance, and not only AmazonEC2. The following command line distributes between an instance on your localhost, two instances on server sv1 and 2 on an AmazonEC2 instance with key mykey.pem.

```
opa-cloud hello_chat.exe --host-pem ubuntu@ec2-XXX.amazonaws.com,2 --pem mykey.pem --host  ↩
    localhost --host sv1,2
```

You can, of course, use Amazon's load-balancer instead of Opa's load balancer.

WARNING Check that each instance can reach every other one e.g. launching EC2 instances from a computer in a local network may raise issues because the EC2 instances won't be able to reach hosts in the sub-network.

7.5.8 My application does not use any database, do I have to get it started anyway?

For some reason, you may not want to start the Opa database server e.g. your service may not need one. You can specify it in the command line with the *--no-db* option:

```
opa-cloud --host localhost,3 myapp.exe --nodb
```

7.5.9 Can I specify different public keys for different hosts?

No. At the time, `opa-cloud` does not offer this flexibility. Although we are working to implement it because we believe it would be a great feature.

7.5.10 Port 8080 (default port) is not available, how can I change it?

The load-balancer is set to listen on port 8080, by default. You can change that with option *--port*:

```
opa-cloud --host localhost,3 myapp.exe --port 2501
```

7.5.11 How can I check whether I have HAProxy installed or not?

Try the following command:

```
$ haproxy -v
HA-Proxy version 1.3.20 2009/08/09
Copyright 2000-2009 Willy Tarreau <w@1wt.eu>

$
```

Once again, don't worry about the version number. Although, the latest the better, you will be able to load-balance with older versions.

7.5.12 I want to use my own version of HAProxy, can I specify it to `opa-cloud`?

Yes. `opa-cloud` offers an option to do that:

```
opa-cloud --host localhost,4 myapp.exe --haproxy /path/to/hapoxy
```

7.5.13 What else ?

`opa-cloud` can do a bit more then what you've seen so far. It offers several command-line option for a better control of your distribution:

```
opa-cloud --help
```

Chapter 8

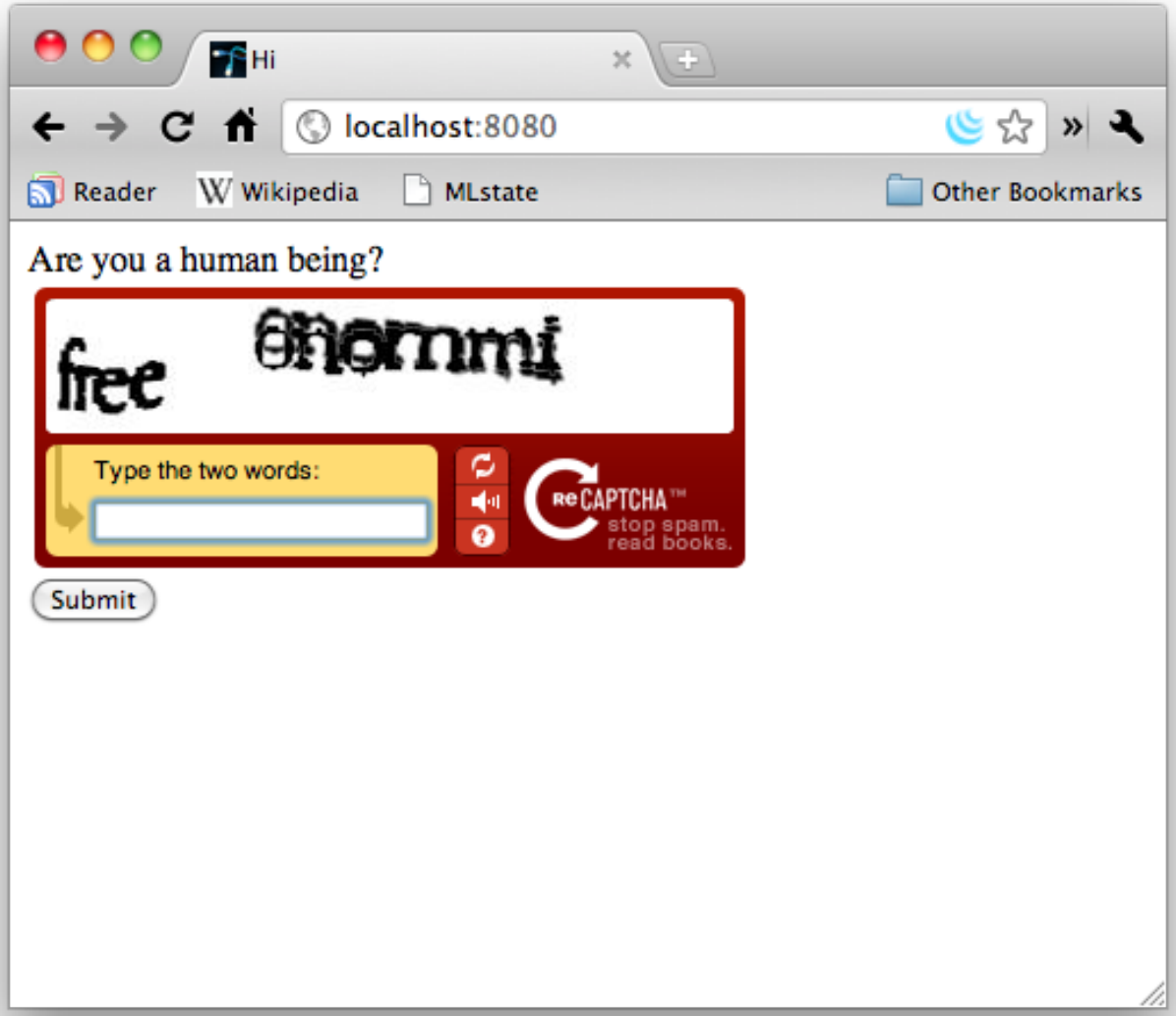
Hello, reCaptcha (and the rest of the world)

In this chapter, we will see how to plug an external API to the Opa platform — here, Google reCaptcha, an API used to protect forms against spammers by attempting to determine whether the person filling the form is actually a human being. Along the way, we will introduce the Opa Binding System Library (or BSL) and some of the mechanisms provided by Opa to permit modular, safe programming.

As we will be interacting with JavaScript, some notions of the JavaScript syntax are useful to understand this chapter.

8.1 Overview

As in previous chapters, let us start with a picture of the application we will develop in this chapter:



This simple application prompts users to recognize words that are considered too difficult to read by a computer in the current state of the art of character recognition — typically because Google's own resources have failed to make sense of these words — and determines that a user is indeed a human being if the answer corresponds to that of sufficient other users through the world. The feature is provided by Google, as an API called reCaptcha. This API involves some code that must be executed on the client (to display the user interface and offer some interactivity) and some code that must be executed on the server (to contact Google's servers and check the validity of the answer).

If you are curious, this is the full source code of our application:

```
import tutorial.recaptcha

/**
 * A test configuration for the reCaptcha, based on a public/private key pair
 * registered for "example.com".
 */
@server_private config = {
 private = {
   privkey = "6LdVTcQSAAAAAJGuyHbunFigtJ1HuI1nkVtwAahg"
 }
 public  = {
```

```
   pubkey =  "6LdVTcQSAAAAABMAPRuRawMTsX0vfW2O2c3bDeeL"
   theme  = {some = "red"}
 }
}

/**
 * After validation, just display the result.
 */
after_validation =
  | {success = _} -> Dom.transform([#status <- <>success</>])
  | {failure = _} -> Dom.transform([#status <- <>failure</>])

/**
 * Main application.
 */
server = one_page_server("Hi", ->
  (implementation, recaptcha) = Recaptcha.make(config)
  <div>
     Are you a human being?<br />
     {recaptcha}
     <button onclick={_ -> Recaptcha.validate(implementation, after_validation)} > Submit  ↩
         </button>
     <div id=#status></div>
  </div>
)
```

Run

Of course, since the features are provided by Google reCaptcha, the most interesting aspects of the code are not to be found in the source of the application itself, but in how we define the binding of reCaptcha for Opa.

This is done in two parts. At high-level, we find the Opa package:

```
package tutorial.recaptcha
import stdlib.web.client

/**
 * A user interface component based on Google reCaptcha, used to differenciate between
 * human beings and spam/link harvest bots by confronting users to read words from scanned
 * books.
 *
 * This module implements a very thin layer on top of the Google reCaptcha AJAX API.
 * For more information on reCaptcha API, see {{:http://code.google.com/apis/recaptcha/ ↩
    intro.html} the official
 * documentation}.
 */

/**
 * {1 Types defined in this module}
 */

/**
 * {2 Setting up the recaptcha}
 */

/**
 * A configuration for the captcha.
 */
type Recaptcha.config =
{
   /**
    * Private fields, isolated for clarity: for security reasons, content of these fields
    * must never be sent to the client.
```

```
    */
   private:
   {
      /**
       * The private access key to the service.
       *
       * Obtain one (and the corresponding public key) from {{:https://www.google.com/ ↩
    recaptcha/admin/create} reCaptcha}
       */
      privkey: string
   }
   /**
    * Public fields, isolated for clarity: the content of these fields can be shared with
    * the client without causing any security issue.
    */
   public:
   {
      /**
       * The public access key to the service.
       *
       * Obtain one (and the corresponding private key) from {{:https://www.google.com/ ↩
    recaptcha/admin/create} reCaptcha}
       */
      pubkey:  string

      /**
       * Name of the visual theme to use for this captcha.
       *
       * If [{none}], we default to the "red" style. Other acceptable styles are: "white",  ↩
    "blackglass",
       * "clean" or "custom". For more details, see the {{:http://code.google.com/apis/ ↩
    recaptcha/docs/customization.html}
       * reCaptcha customization manual}.
       */
      theme:   option(string)
   }
}

/**
 * {2 The reCaptcha object}
 */

/**
 * An abstract object implementing the methods of the reCaptcha.
 */
@abstract type Recaptcha.implementation = {
     /**Place a request to the reCaptcha server to verify that user entry is correct.
        @param challenge
        @param response
        @param callback*/
     validate: (string, string, (Recaptcha.result -> void) -> void)
     reload:   -> void /**Reload the reCaptcha, displaying a different challenge*/
     destroy:  -> void /**Destroy the reCaptcha*/
}


/**
 * {2 Manipulation of recapatcha results}
 */
type Recaptcha.success = {captcha_solved} /**The captcha is correct.*/
type Recaptcha.failure =
     {captcha_not_reachable: WebClient.failure} /**Could not reach the distant server.*/
```

```
 /   {upstream: string}      /**Upstream error, documented but left unspecified by Google ↩
     .*/
 /   {unknown: list(string)} /**Server could be reached, but produced an error that doesn't ↩
      match the specifications provided by Google.
                                Possible cause: proxy problem.*/
 /   {empty_answer}          /**Recaptcha guidelines mention that we should never send  ↩
     answers that are empty.*/

type Recaptcha.result = {success: Recaptcha.success} / {failure: Recaptcha.failure}


/**
 * {1 Implementation}
 */

Recaptcha =
{{
   /**
    * Construct a reCaptcha.
    *
    * Note: The API provided by Google only permits one reCaptcha per page.
    *
    * @param config The configuration for the recaptcha. Note that, as this configuration
    * contains security-sensitive information, it {e must} be stored on the server.
    *
    * @return An object representing the methods of the reCaptcha, which you can later use
    * to validate an entry, reload the reCaptcha, etc. and a [xhtml] extract which should  ↩
    be
    * inserted in a page to display the captcha dialog.
    */
   make(config: Recaptcha.config): (Recaptcha.implementation, xhtml) =
   (
     id = Dom.fresh_id()
     xhtml = <div id={id} onready={_ -> onready(id, config.public.pubkey, config.public. ↩
         theme?"red")}/>
     (make_implementation(config.private.privkey), xhtml)
   )

   /**
    * Attempt to determine whether a user is a human being.
    *
    * This request calls upon Google's reCaptcha servers to determine whether
    * the user is probably a human being.
    *
    * @param implementation A reCaptcha.
    * @param callback A function which will be called once the server has provided an  ↩
    answer
    */
   validate(implementation: Recaptcha.implementation, callback: Recaptcha.result -> void):  ↩
       void =
   (
      ~{challenge response} = get_token()
      implementation.validate(challenge, response, callback)
   )

   /**
    * Display a new challenge
    *
    * @param implementation A reCaptcha.
    */
   reload(implementation: Recaptcha.implementation): void =
   (
```

```
  implementation.reload()
)

/**
 * Clean-up and remove the reCaptcha.
 *
 * If you are using the reCaptcha in a dialog box, it is generally a good idea to call
 * this function if the user closes the box.
 *
 * @param implementation A reCaptcha.
 */
destroy(implementation: Recaptcha.implementation): void =
(
  implementation.destroy()
)

/**
 * {2 Private implementation details}
 */

/**
 * The URIs for the API
 */

@private path_validate_uri = Option.get(Parser.try_parse(Uri.uri_parser, "http://www. ↩
    google.com/recaptcha/api/verify"))
@private path_js_uri       = Option.get(Parser.try_parse(Uri.uri_parser, "http://www. ↩
    google.com/recaptcha/api/js/recaptcha_ajax.js"))

/**
 * Perform client-side delayed initialization.
 *
 * Load the reCaptcha script if necessary, then perform script initialization.
 *
 * @param id The dom identifier of the xhtml component which will contain the reCaptcha.
 * @param pubkey The public key for this reCaptcha.
 * @param theme The name of the theme for this reCaptcha (in case of doubt, "red" is a  ↩
 good choice).
 */
@private onready(id: string, pubkey: string, theme: string): void =
(
   Client.Script.load_uri_then(path_js_uri, ->
           (%% Recaptcha.init %%)(id, pubkey, theme)
   )
)

@private client_reload = %%Recaptcha.reload%% /**Implementation of [reload]*/
@private client_destroy = %%Recaptcha.destroy%% /**Implementation of [destroy]*/

/**
 * Construct the object representing the reCaptcha
 *
 * @param privkey The private key.
 * @return An object containing the necessary methods to contact the reCaptcha server.
 */
@private make_implementation(privkey: string): Recaptcha.implementation =
(
   validate(challenge, response, callback:Recaptcha.result -> void) =
   (
     //By convention, do not even post a request if the data is empty
     if String.is_empty(challenge) || String.is_empty(response) then callback({failure = ↩
         {empty_answer}})
```

```
        else
        (
          /**POST request, formatted as per API specifications*/
          data = [("privatekey", privkey),
                  ("remoteip",   "{HttpRequest.get_ip()?(127.0.0.1)}"),
                  ("challenge",  challenge),
                  ("response",   response)]
          /**Handle POST failures, decode reCaptcha responses, convert this to [reCaptcha. ↩
              result].*/
          with_result =
          (
            | ~{failure} -> callback({failure = {captcha_not_reachable = failure}})
            | ~{success} ->
              details = String.explode("\n", success.content)
              match details with
                | ["true" | _]        -> callback({success = {captcha_solved}})
                | ["false", code | _] -> callback({failure = {upstream = code}})
                | _ -> callback({failure = {unknown = details}})
          )
          /**Encode arguments, POST them*/
          WebClient.Post.try_post_with_options_async(path_validate_uri,
                WebClient.Post.of_form({WebClient.Post.default_options with content = {some ↩
                     = data}}),
                with_result)

        )
      )
      {~validate reload=client_reload destroy=client_destroy}
  )

  /**
   * Grab all necessary information from the user interface.
   */
  @private get_token(): {challenge:string response:string} =
  (
    result = ({challenge = (%%Recaptcha.get_challenge%%)()
               response  = (%%Recaptcha.get_response%%)()})
    do (%%Recaptcha.destroy%%)()
    result
  )
}}
```

At low level, we find the JavaScript binding:

```
##register init: string, string, string -> void
##args(id, pubkey, theme)
{
  Recaptcha.create(pubkey,
      id,
      {
        theme: theme,
        callback: Recaptcha.focus_response_field
      }
  );
}

##register get_challenge: -> string
##args()
{
    return (Recaptcha.get_challenge()||"")
}
```

```
##register get_response: -> string
##args()
{
    return (Recaptcha.get_response()||"")
}

##register destroy: -> void
##args()
{
    Recaptcha.destroy();
}

##register reload: -> void
##args()
{
    Recaptcha.reload();
}
```

In the rest of the chapter, we will walk you through all the concepts and constructions introduced in these listings.

8.2 Populating the BSL

The *documentation* of the reCaptcha API details five methods of a JavaScript object called `Recaptcha`, that should be called at distinct stages of the use of the API. Our first step will therefore be to bind each of these methods to Opa, through the mechanism of the Binding System Library.

About the BSL
The Binding System Library was developed to allow *binding* external features to Opa. The mechanism is used at low-level in Opa itself, to bind Opa to browser features, but also to server features.
At the time of this writing, the BSL provides native bindings with JavaScript and OCaml, and can be used to bind to most languages, including C.
This chapter shows how to plug Opa with some Javascript client code. The chapter about the BSL will also show how to bind Opa and Ocaml, and how to use C primitives from Opa.

For this purpose, we will create a file called `recaptcha.js`, and that we will populate with Opa-to-JavaScript bindings.

Let us start with initialization. The documentation states that any use of the JavaScript API must initialize the `Recaptcha` object as follows:

```
Recaptcha.create(pubkey,
   id,
   {
      theme: theme,
      callback: Recaptcha.focus_response_field
   }
);
```

where `pubkey` is a public key obtained from the reCaptcha admin site, `id` is the identifier of the UI component that will host the reCaptcha and `theme` is the name of the visual theme.

For our purpose, as we are binding the API, these three values should be function arguments, for a function accepting three character strings and returning nothing meaningful.

We specify this as follows:

```
##register init: string, string, string -> void
##args(id, pubkey, theme)
{
```

```
  Recaptcha.create(pubkey,
      id,
      {
        theme: theme,
        callback: Recaptcha.focus_response_field
      }
  );
}
```

The first line registers a function called `init` (we could have called it `create`, as the JavaScript method, but `init` fits better with the Opa conventions). This function is implemented in JavaScript and its type is specified as `string, string, string -> void`.

The second line gives names to arguments, respectively `id`, `pubkey` and `theme`. If you are familiar with JavaScript, you can think of BSL keyword `##args` as a counterpart to `function`, with stricter checks.

The rest of the extract is regular JavaScript, copied directly from the documentation of reCaptcha.

About void
The only surprise may be that there is no final `return`. Indeed, in regular JavaScript, functions with no `return` value actually return `undefined`. By opposition, Opa is stricter and does not allow `undefined` values. Since our definition states that `init` always returns a `void`, the BSL will apply an automatic transformation of the Javascrit code so that once bined in Opa, the function returns `void`.
In Opa, functions always return a value, even if this value is `void`. This is true even of functions implemented in JavaScript.
Therefore, if a BSL JavaScript function has type `... -> void`, once in Opa, it returns `void`.

The rest of the API is quite similar. Functions `reload` and `destroy`, which serve respectively to display a new challenge or to clean the screen once the reCaptcha has become useless, are bound as follows:

```
##register reload: -> void
##args()
{
    Recaptcha.reload();
}

##register destroy: -> void
##args()
{
    Recaptcha.destroy();
}
```

These bindings should not surprise you. Simply note that we write `##args()` if a function does not take any argument.

Binding functions `get_challenge` and `get_response` is quite similar. The first of these functions returns an opaque string that can be used by the reCaptcha server to determine which image has been sent to the user. The second returns the text entered by the user.

```
##register get_challenge: -> string
##args()
{
    return (Recaptcha.get_challenge()||"")
}

##register get_response: -> string
##args()
{
    return (Recaptcha.get_response()||"")
}
```

Note that we do not return simply `Recaptcha.get_challenge()` or `Recaptcha.get_response()`. Indeed, experience with the reCaptcha API shows that, in some (undocumented) cases, these functions return value `null`, which is definitely not a valid Opa `string`. For this purpose, we normalize the `null` value to the empty string `""`.

About null
In Opa, the JavaScript value `null` is meaningless. An Opa function implemented as JavaScript and which returns `null` (or an object in which some fields are `null`) is a programmer error.

With this, the source code for the BSL bindings is complete. Before proceeding to the Opa side, we just need to compile this source code:

```
opa-plugin-builder recaptcha.js -o recaptcha
```

For more details about *opa-plugin-builder*, you can refer to its documentation.

We are now done with JavaScript.

8.3 Typing the API

The next step is to connect the BSL to the server component and wrap the features as a nice Opa API.

Looking at the documentation of reCaptcha, we may see that a reCaptcha accepts exactly three meaningful arguments:

- a private key, which we need to obtain manually from reCaptcha, and which should never be transmitted to the client, for security reasons;
- a public key, which we obtain along with the private key;
- an optional theme name.

We group these arguments as a record type, as follows:

```
type Recaptcha.config =
{
   private:
   {
      privkey: string
   }
   public:
   {
      pubkey:  string
      theme:   option(string)
   }
}
```

By convention, records which simply serve to group arguments under meaningful names are called *configurations* and their name ends with `.config`. Here, in order to avoid confusions, we have split this record in two subrecords, called respectively `private`, for information that we want to keep on the server, and `public`, for information that can leave the client without breaching security.

Also, looking at the documentation of the reCaptcha server-side API, we find out that communications with the reCaptcha server can yield the following results:

- a success;
- a failure, which may either mean that the user failed to identify the text, or that some other issue took place, including a communication error between Google servers.

Two additional error cases may appear:

- a communication error between your application and Google (typically, due to a network outage);
- a message returned by Google which does not match the documentation (unlikely but possible);
- an empty answer provided by the user, in which case communication should not even take place.

While these distinct results are represented as strings in the API, in Opa, we will prefer a sum type, which we define as follows:

```
type Recaptcha.success = {captcha_solved} /**The captcha is correct.*/
type Recaptcha.failure =
     {captcha_not_reachable: WebClient.failure} /**Could not reach the distant server.*/
 /   {upstream: string}      /**Upstream failure. Could be a user error, but the code is ↩
     not meant to be exploited.*/
 /   {unknown: list(string)} /**Server could be reached, but produced an error that doesn't ↩
      match the specifications provided by Google.
                                Possible cause: proxy problem.*/
 /   {empty_answer}          /**Recaptcha guidelines mention that we should never send ↩
     answers that are empty.*/

type Recaptcha.result = {success: Recaptcha.success} / {failure: Recaptcha.failure}
```

And finally, we define one last type, that of the reCaptcha *implementation*, as follows:

```
type Recaptcha.implementation = {
     reload:   -> void /**Reload the reCaptcha, displaying a different challenge*/
     destroy:  -> void /**Destroy the reCaptcha*/
     /**Place a request to the reCaptcha server to verify that user entry is correct.
        @param challenge
        @param response
        @param callback*/
     validate: (string, string, (Recaptcha.result -> void) -> void)
}
```

The role of this type is to encapsulate the functions of the reCaptcha after construction.

This type offers three fields. The first two will map respectively to the `reload` method we have bound earlier and to the `destroy` method we have bound earlier. The third, `validate`, is a more powerful function whose role will be to send to the reCaptcha server the challenge, the response entered by the user, to wait for the server response and to trigger a function with the result.

Objects in Opa
Opa is not an Object-Oriented Programming Language in the traditional sense of the term. However, it is a *Higher-Order Programming Language*, which means among other things that all the major features of Object-Oriented Programming can be found in Opa, and more.
In our listing, values of type `Recaptcha.implementation` contain fields, some of which are functions — not unlike Objects in OO languages may contain fields and methods.
Although this is a slight misues of the term, we often call such records, that is records containing fields, some of which are functions, *Objects*.

Now that the types are ready, we can write the functions that manipulate them.

8.4 Implementing the API

The documentation of reCaptcha mentions a JavaScript library, provided by reCaptcha, which needs to be loaded prior to initializing the reCaptcha. We handle this in a function `onready` which we will use shortly:

```
onready(id: string, pubkey: string, theme: string): void =
(
   Client.Script.load_uri_then(path_js_uri,
      -> (%% Recaptcha.init %%)(id, pubkey, theme)
   )
)
```

This function makes use of `Client.Script.load_uri_then`, a function provided by the library to load a distant script and, once loading is complete, to invoke a second function. First argument, `path_js_uri`, is a constant representing the URI at which the JS is available. We will define it a bit later. The second argument is a function that makes use of a construction you have never seen:

```
(%% Recaptcha.init %%)
```

This is simply function `init`, as we defined it a few minutes ago in JavaScript.

Calling the BSL
To use an Opa value defined in the BSL (that is, in JavaScript, C, OCaml, or any other language), the syntax is

```
(%% NameOfThePlugin.name_of_the_value %%)
```

Replace `NameOfThePlugin` by the name of the file you have passed to `opa-plugin-builder` and `name_of_the_value` by the name you have registered with `##register`. Capitalization of plug-in names is ignored. The contains between the 2 *%%* is called the *key* of the external primitive. A part of the binding system reference details how keys are associated to primitives.

In other words, this function `onready` loads the JavaScript provided by Google, then initializes the reCaptcha. To call this function, we will make use of a `Recaptcha.config` and a `ready` event, to ensure that it is called only on a browser that actually makes use of the reCaptcha. We will need this initialization in our constructor for `Recaptcha.implementation`.

Multiple loads
Module `Client.Script` provides several functions for loading JavaScript. These functions ensure that a JavaScript URI will only be loaded once. In other words, you do not have to worry about the same JavaScript being loaded and executed multiple times.

The second important function, which we will also need to build our `Recaptcha.implementation`, is the validation. We implement it as follows:

```
validate(challenge, response, callback:Recaptcha.result -> void): void =
(
  //By convention, do not even post a request if the data is empty
  if String.is_empty(challenge) || String.is_empty(response) then callback({failure = { ↩
      empty_answer}})
  else
  (
    /**POST request, formatted as per API specifications*/
    data = [("privatekey", privkey),
            ("remoteip",   "{HttpRequest.get_ip()?(127.0.0.1)}"),
            ("challenge",  challenge),
            ("response",   response)]
    /**Handle POST failures, decode reCaptcha responses, convert this to [reCaptcha.result ↩
        ].*/
    with_result =
    (
      | ~{failure} -> callback({failure = {captcha_not_reachable = failure}})
      | ~{success} ->
```

```
        details = String.explode("\n", success.content)
        match details with
          | ["true" | _]        -> callback({success = {captcha_solved}})
          | ["false", code | _] -> callback({failure = {upstream = code}})
          | _ -> callback({failure = {unknown = details}})
    )
        /**Encode arguments, POST them*/
    WebClient.Post.try_post_with_options_async(path_validate_uri,
        WebClient.Post.of_form({WebClient.Post.default_options with content = {some =  ↩
            data}}),
        with_result)
)
```

Although this listing uses several functions that you have not seen yet, it should not surprise you. The first part responds immediately if the challenge or the response is negative. This complies with the specification of reCaptcha, in addition to saving precious resources on your server. We then build `data`, a list of association of the arguments expected by the reCaptcha API, and `with_result`, a function that handles the return of our `{post}` request by analyzing the resulting `string`. Note the use of function `WebClient.Post.of_form`, which converts a request using a list of associations, as are built by HTML forms, into a raw, `string`-based request. Also note function `String.explode`, which splits a string in a list of substrings.

From `validate`, as well as our JavaScript implementations of `reload` and `destroy`, we may now construct our `Recaptcha.impl` as follows:

```
client_reload = %%Recaptcha.reload%%       /**Implementation of [reload]*/
client_destroy = %%Recaptcha.destroy%%      /**Implementation of [destroy]*/
make_implementation(privkey: string): Recaptcha.implementation =
(
   validate(challenge, response, callback:Recaptcha.result -> void) =
   (
     //By convention, do not even post a request if the data is empty
     if String.is_empty(challenge) || String.is_empty(response) then callback({failure = { ↩
         empty_answer}})
     else
     (
    /**POST request, formatted as per API specifications*/
       data = [("privatekey", privkey),
               ("remoteip",   "{HttpRequest.get_ip()?(127.0.0.1)}"),
               ("challenge",  challenge),
               ("response",   response)]
       /**Handle POST failures, decode reCaptcha responses, convert this to [reCaptcha. ↩
           result].*/
       with_result =
       (
      | ~{failure} -> callback({failure = {captcha_not_reachable = failure}})
        | ~{success} ->
          details = String.explode("\n", success.content)
          match details with
            | ["true" | _]        -> callback({success = {captcha_solved}})
            | ["false", code | _] -> callback({failure = {upstream = code}})
            | _ -> callback({failure = {unknown = details}})
       )
       /**Encode arguments, POST them*/
       WebClient.Post.try_post_with_options_async(path_validate_uri,
          WebClient.Post.of_form({WebClient.Post.default_options with content = {some =  ↩
              data}}),
          with_result)

     )
   )
   {~validate reload=client_reload destroy=client_destroy}
)
```

With this function, we implement a new function `make`, to construct both the `Recaptcha.implementation` and the user interface `xhtml` component that connects to this implementation:

```
make(config: Recaptcha.config): (Recaptcha.implementation, xhtml) =
(
  id = Dom.fresh_id()
  xhtml = <div id={id} onready={_ -> onready(id, config.public.pubkey, config.public.theme ↩
      ?"red")}/>
  (make_implementation(config.private.privkey), xhtml)
)
```

One more utility function will be useful, to read the user interface and clean it up immediately:

```
get_token(): {challenge:string response:string} =
(
  result = ({challenge = (%%Recaptcha.get_challenge%%)()
             response  = (%%Recaptcha.get_response%%)()})
  do (%%Recaptcha.destroy%%)()
  result
)
```

With this function, we now have exposed all the features we need to use a reCaptcha. However, at this stage, we are exposing a great deal of the implementation. Consequently, our next step will be to make the implementation abstract and to encapsulate the features in a module.

8.5 Modularizing the features

It is generally a good idea to split large (or even small) projects into *packages*, as follows:

```
package tutorial.recaptcha
```

About packages
In Opa, a *package* is a unit of compilation and abstraction. Packages can be distributed separately as compiled code, packages can hold private values, abstract types, etc.
The following declaration states that the current file is part of package `package_name`:

```
package package_name
```

Conversely, the following declaration states that the current file makes use of package `package_name`:

```
import package_name
```

Generally, in Opa, packages are assembled into package hierarchies, using reverse domain notation.

Now that we have placed our code in a package, we may decide that some of our types are *abstract*, as follows:

```
@abstract type Recaptcha.implementation = {
     validate: (string, string, (Recaptcha.result -> void) -> void)
     reload:   -> void /**Reload the reCaptcha, displaying a different challenge*/
     destroy:  -> void /**Destroy the reCaptcha*/
}
```

About abstract types
An *abstract* type is a type whose *definition* can only be used in the package in which it was defined, although its *name* might be used outside of the package.
This feature provides a very powerful mechanism for avoiding errors and ensuring that data invariants remain unbroken, but also to ensure that third-party developers do not base their work on characteristics that may change at a later stage.
Consider the following example:

```
package example

@abstract type cost = float
```

In this example, type `cost` is defined as a synonym of `float`. In package `example`, any `float` can be used as a `cost` and reciprocally. However, outside of package `example`, `cost` and `float` are two distinct types. In particular, we have ensured that only the code of package `example` can create new values of type `cost`. If our specifications expect that a `cost` is always strictly positive, we have succesfully restriced the perimeter of the code we need to check to ensure that this invariant remains unbroken: only package `example` needs to be verified.

With this change, methods `validate`, `destroy` and `reload` can now be called only from our package. As these methods offer important features, we certainly wish to provide some form of access to the methods, as follows:

```
validate(implementation: Recaptcha.implementation, callback: Recaptcha.result -> void):  ↩
    void =
(
   ~{challenge response} = get_token()
   implementation.validate(challenge, response, callback)
)

reload(implementation: Recaptcha.implementation): void =
(
  implementation.reload()
)

destroy(implementation: Recaptcha.implementation): void =
(
  implementation.destroy()
)
```

Note construction

```
~{challenge response} = get_token()
```

This is a simple case of pattern-matching, and it is equivalent to:

```
x = get_token()
challenge = x.challenge
response  = x.response
```

Actually, this pattern-matching is generally a little faster.

For further modularization, we will group all our functions as a *module*, and take the opportunity to hide the function and constants that should not be called from the outside of the module:

```
Recaptcha =
{{
/**
 * {1 Public functions}
 */
   make(config: Recaptcha.config): (Recaptcha.implementation, xhtml) =
   (
     id = Dom.fresh_id()
```

```
  xhtml = <div id={id} onready={_ -> onready(id, config.public.pubkey, config.public. ↩
      theme?"red")}/>
  (make_implementation(config.private.privkey), xhtml)
)

validate(implementation: Recaptcha.implementation, callback: Recaptcha.result -> void): ↩
    void =
(
   (challenge, response) = get_token()
   implementation.validate(challenge, response, callback)
)

reload(implementation: Recaptcha.implementation): void =
(
  implementation.reload()
)

destroy(implementation: Recaptcha.implementation): void =
(
  implementation.destroy()
)

/**
 * {1 Private implementation details}
 */

/**
 * The URIs for the API
 */

@private path_validate_uri = Option.get(Parser.try_parse(Uri.uri_parser, "http://www. ↩
    google.com/recaptcha/api/verify"))
@private path_js_uri       = Option.get(Parser.try_parse(Uri.uri_parser, "http://www. ↩
    google.com/recaptcha/api/js/recaptcha_ajax.js"))

@private onready(id: string, pubkey: string, theme: string): void =
(
   Client.Script.load_uri_then(path_js_uri, ->
           (%% Recaptcha.init %%)(id, pubkey, theme)
   )
)

@private client_reload = %%Recaptcha.reload%% /**Implementation of [reload]*/
@private client_destroy = %%Recaptcha.destroy%% /**Implementation of [destroy]*/

@private make_implementation(privkey: string): Recaptcha.implementation =
(
   validate(challenge, response, callback:Recaptcha.result -> void) =
   (
     //By convention, do not even post a request if the data is empty
     if String.is_empty(challenge) || String.is_empty(response) then callback({failure = ↩
         {empty_answer}})
     else
     (
       /**POST request, formatted as per API specifications*/
       data = [("privatekey", privkey),
               ("remoteip",   "{HttpRequest.get_ip()?(127.0.0.1)}"),
               ("challenge",  challenge),
               ("response",   response)]
       /**Handle POST failures, decode reCaptcha responses, convert this to [reCaptcha. ↩
           result].*/
       with_result =
```

```
          (
            | ~{failure} -> callback({failure = {captcha_not_reachable = failure}})
            | ~{success} ->
              details = String.explode("\n", success.content)
              match details with
                | ["true" | _]        -> callback({success = {captcha_solved}})
                | ["false", code | _] -> callback({failure = {upstream = code}})
                | _ -> callback({failure = {unknown = details}})
          )
          /**Encode arguments, POST them*/
          WebClient.Post.try_post_with_options_async(path_validate_uri,
                WebClient.Post.of_form({WebClient.Post.default_options with content = {some ↩
                     = data}}),
                with_result)

        )
      )
      {~validate reload=client_reload destroy=client_destroy}
   )

   @private get_token(): (string, string) =
   (
     result = ({challenge = (%%Recaptcha.get_challenge%%)()
                response  = (%%Recaptcha.get_response%%)()})
     do (%%Recaptcha.destroy%%)()
     result
   )
}}
```

About modules
A *module* is a form of record introduced with `{{ ... }}` instead of `{ ... }`.
Modules offer a few syntactic enrichments: - any field may be declared as `@private`, which forbids from using it outside of the module; - fields can be declared in any order, even if they have dependencies (including circular dependencies).
In addition, modules are typed slightly differently from regular records. We will detail this in another chapter.

And with this, our binding is complete.

We may compile our package with

```
opa recaptcha.opp hello_recaptcha.opa
```

Let us recapitulate the Opa source code:

```
package tutorial.recaptcha
import stdlib.web.client

/**
 * A user interface component based on Google reCaptcha, used to differenciate between
 * human beings and spam/link harvest bots by confronting users to read words from scanned
 * books.
 *
 * This module implements a very thin layer on top of the Google reCaptcha AJAX API.
 * For more information on reCaptcha API, see {{:http://code.google.com/apis/recaptcha/ ↩
    intro.html} the official
 * documentation}.
 */

/**
 * {1 Types defined in this module}
```

```
 */

/**
 * {2 Setting up the recaptcha}
 */

/**
 * A configuration for the captcha.
 */
type Recaptcha.config =
{
   /**
    * Private fields, isolated for clarity: for security reasons, content of these fields
    * must never be sent to the client.
    */
   private:
   {
      /**
       * The private access key to the service.
       *
       * Obtain one (and the corresponding public key) from {{:https://www.google.com/ ↩
    recaptcha/admin/create} reCaptcha}
       */
      privkey: string
   }
   /**
    * Public fields, isolated for clarity: the content of these fields can be shared with
    * the client without causing any security issue.
    */
   public:
   {
      /**
       * The public access key to the service.
       *
       * Obtain one (and the corresponding private key) from {{:https://www.google.com/ ↩
    recaptcha/admin/create} reCaptcha}
       */
      pubkey:  string

      /**
       * Name of the visual theme to use for this captcha.
       *
       * If [{none}], we default to the "red" style. Other acceptable styles are: "white",  ↩
    "blackglass",
       * "clean" or "custom". For more details, see the {{:http://code.google.com/apis/ ↩
    recaptcha/docs/customization.html}
       * reCaptcha customization manual}.
       */
      theme:   option(string)
   }
}

/**
 * {2 The reCaptcha object}
 */

/**
 * An abstract object implementing the methods of the reCaptcha.
 */
@abstract type Recaptcha.implementation = {
     /**Place a request to the reCaptcha server to verify that user entry is correct.
        @param challenge
```

```
        @param response
        @param callback*/
     validate: (string, string, (Recaptcha.result -> void) -> void)
     reload:   -> void /**Reload the reCaptcha, displaying a different challenge*/
     destroy:  -> void /**Destroy the reCaptcha*/
}


/**
 * {2 Manipulation of recapatcha results}
 */
type Recaptcha.success = {captcha_solved} /**The captcha is correct.*/
type Recaptcha.failure =
     {captcha_not_reachable: WebClient.failure} /**Could not reach the distant server.*/
 /   {upstream: string}      /**Upstream error, documented but left unspecified by Google ↩
     .*/
 /   {unknown: list(string)} /**Server could be reached, but produced an error that doesn't ↩
      match the specifications provided by Google.
                                Possible cause: proxy problem.*/
 /   {empty_answer}          /**Recaptcha guidelines mention that we should never send  ↩
     answers that are empty.*/

type Recaptcha.result = {success: Recaptcha.success} / {failure: Recaptcha.failure}


/**
 * {1 Implementation}
 */

Recaptcha =
{{
   /**
    * Construct a reCaptcha.
    *
    * Note: The API provided by Google only permits one reCaptcha per page.
    *
    * @param config The configuration for the recaptcha. Note that, as this configuration
    * contains security-sensitive information, it {e must} be stored on the server.
    *
    * @return An object representing the methods of the reCaptcha, which you can later use
    * to validate an entry, reload the reCaptcha, etc. and a [xhtml] extract which should  ↩
    be
    * inserted in a page to display the captcha dialog.
    */
   make(config: Recaptcha.config): (Recaptcha.implementation, xhtml) =
   (
     id = Dom.fresh_id()
     xhtml = <div id={id} onready={_ -> onready(id, config.public.pubkey, config.public. ↩
         theme?"red")}/>
     (make_implementation(config.private.privkey), xhtml)
   )

   /**
    * Attempt to determine whether a user is a human being.
    *
    * This request calls upon Google's reCaptcha servers to determine whether
    * the user is probably a human being.
    *
    * @param implementation A reCaptcha.
    * @param callback A function which will be called once the server has provided an  ↩
    answer
    */
```

```
validate(implementation: Recaptcha.implementation, callback: Recaptcha.result -> void):  ↩
    void =
(
   ~{challenge response} = get_token()
   implementation.validate(challenge, response, callback)
)

/**
 * Display a new challenge
 *
 * @param implementation A reCaptcha.
 */
reload(implementation: Recaptcha.implementation): void =
(
  implementation.reload()
)

/**
 * Clean-up and remove the reCaptcha.
 *
 * If you are using the reCaptcha in a dialog box, it is generally a good idea to call
 * this function if the user closes the box.
 *
 * @param implementation A reCaptcha.
 */
destroy(implementation: Recaptcha.implementation): void =
(
  implementation.destroy()
)

/**
 * {2 Private implementation details}
 */

/**
 * The URIs for the API
 */

@private path_validate_uri = Option.get(Parser.try_parse(Uri.uri_parser, "http://www. ↩
    google.com/recaptcha/api/verify"))
@private path_js_uri       = Option.get(Parser.try_parse(Uri.uri_parser, "http://www. ↩
    google.com/recaptcha/api/js/recaptcha_ajax.js"))

/**
 * Perform client-side delayed initialization.
 *
 * Load the reCaptcha script if necessary, then perform script initialization.
 *
 * @param id The dom identifier of the xhtml component which will contain the reCaptcha.
 * @param pubkey The public key for this reCaptcha.
 * @param theme The name of the theme for this reCaptcha (in case of doubt, "red" is a  ↩
 good choice).
 */
@private onready(id: string, pubkey: string, theme: string): void =
(
   Client.Script.load_uri_then(path_js_uri, ->
           (%% Recaptcha.init %%)(id, pubkey, theme)
   )
)

@private client_reload = %%Recaptcha.reload%% /**Implementation of [reload]*/
@private client_destroy = %%Recaptcha.destroy%% /**Implementation of [destroy]*/
```

```
  /**
   * Construct the object representing the reCaptcha
   *
   * @param privkey The private key.
   * @return An object containing the necessary methods to contact the reCaptcha server.
   */
  @private make_implementation(privkey: string): Recaptcha.implementation =
  (
     validate(challenge, response, callback:Recaptcha.result -> void) =
     (
       //By convention, do not even post a request if the data is empty
       if String.is_empty(challenge) || String.is_empty(response) then callback({failure = ↩
            {empty_answer}})
       else
       (
         /**POST request, formatted as per API specifications*/
         data = [("privatekey", privkey),
                 ("remoteip",   "{HttpRequest.get_ip()?(127.0.0.1)}"),
                 ("challenge",  challenge),
                 ("response",   response)]
         /**Handle POST failures, decode reCaptcha responses, convert this to [reCaptcha. ↩
             result].*/
         with_result =
         (
           | ~{failure} -> callback({failure = {captcha_not_reachable = failure}})
           | ~{success} ->
             details = String.explode("\n", success.content)
             match details with
               | ["true" | _]        -> callback({success = {captcha_solved}})
               | ["false", code | _] -> callback({failure = {upstream = code}})
               | _ -> callback({failure = {unknown = details}})
         )
         /**Encode arguments, POST them*/
         WebClient.Post.try_post_with_options_async(path_validate_uri,
               WebClient.Post.of_form({WebClient.Post.default_options with content = {some ↩
                    = data}}),
               with_result)

       )
     )
     {~validate reload=client_reload destroy=client_destroy}
  )

  /**
   * Grab all necessary information from the user interface.
   */
  @private get_token(): {challenge:string response:string} =
  (
    result = ({challenge = (%%Recaptcha.get_challenge%%)()
               response  = (%%Recaptcha.get_response%%)()})
    do (%%Recaptcha.destroy%%)()
    result
  )
}}
```

8.6 Testing the API

To test the API, we may write a simple application:

```
import tutorial.recaptcha

/**
 * A test configuration for the reCaptcha, based on a public/private key pair
 * registered for "example.com".
 */
@server_private config = {
 private = {
   privkey = "6LdVTcQSAAAAAJGuyHbunFigtJ1HuI1nkVtwAahg"
 }
 public  = {
   pubkey =  "6LdVTcQSAAAAABMAPRuRawMTsX0vfW2O2c3bDeeL"
   theme  = {some = "red"}
 }
}

/**
 * After validation, just display the result.
 */
after_validation =
  | {success = _} -> Dom.transform([#status <- <>success</>])
  | {failure = _} -> Dom.transform([#status <- <>failure</>])

/**
 * Main application.
 */
server = one_page_server("Hi", ->
  (implementation, recaptcha) = Recaptcha.make(config)
  <div>
     Are you a human being?<br />
     {recaptcha}
     <button onclick={_ -> Recaptcha.validate(implementation, after_validation)} > Submit  ↩
         </button>
     <div id=#status></div>
  </div>
)
```

Run

With the exception of directive `@server_private`, this listing should not surprise you. Here, we placed directive `@server_privat` as a sanity check, to be absolutely certain that the compiler would not duplicate the private key to the client for some optimization. As it turns out, this precaution is not necessary, but reassuring.

Note that the public and private key provided here are registered for domain "example.com". They will work for the example, but should you wish to use the reCaptcha, you should register your own public/private key pair.

We may now compile the application, as usual

```
opa recaptcha.opp hello_recaptcha.opa
```

8.7 Questions

8.7.1 Why an object?

As mentioned, we have defined `Recaptcha.implementation` as an object. This is a good reflex when extending the Opa platform through additional BSL bindings that use data structures can be implemented only on one side.

In Opa, data can be transmitted transparently between the client and the server. This is impossible for data that is meaningful only on the client. This is the case here, as JavaScript object `Recaptcha`, by definition, exists only on the client. However,

wrapping the JavaScript data structure and the functions that manipulate it as an object ensures that the user only ever needs to access methods — and such methods can always be transmitted from the client to the server, making the object data structure side-independent.

Making objects
When a BSL extension to the Opa platform introduces a data structure implemented only on one side, the user *must* never manipulate this data structure directly. *Always* hide this data structure behind an object, whose only fields are functions.

Chapter 9

The core language

While Opa empowers developers with numerous technologies, Opa is first and foremost a programming language. For clarity, the features of the language are presented through several chapters, each dedicated to one aspect of the language. In this chapter, we recapitulate the core constructions of the Opa language, from lexical conventions to data structures and manipulation of data structures, and we introduce a few key functions of the standard library which are closely related to core language constructions.

Read this chapter to find out more about:

- syntax;
- functions;
- records;
- control flow;
- loops;
- patterns and pattern-matching;
- modules;
- text parsers.

Note that this chapter is meant as a reference rather than as a tutorial.

9.1 Lexical convention

Opa accepts standard C/C++/Java/JavaScript-style comments:

```
// one line comment
/*
  multi line comment
*/
/*
  nested
  /* multi line */
  comment
*/
```

A comment is treated as whitespace for all the rules in the syntax that depend on the presence of whitespace.

It is generally a good idea to document values. Documentation can later be collected by the opadoc tool and collected into a cross-referenced searchable document. Documentation takes the place for special comments, starting with `/**`.

```
/**
 * I assure you, this function does lots of useful things!
 * @return 0
**/
zero() = 0
```

Caution
In progress (Soon, a hyperlink to the corresponding chapter)

Ill-formed documentation comments do not break the compilation, they only break the documentation.

9.2 Basic datatypes

Opa has 3 basic datatypes: strings, integers and floating point numbers.

9.2.1 Integers

Integers literals can be written in a number of ways:

```
x = 10 // 10 in base 10
x = 0xA // 10 in base 16, any case works (0Xa, 0XA, Oxa)
x = 0o12 // 10 in base 8
x = 0b1010 // 10 in base 2
```

9.2.2 Floats

Floating point literal can be written in two ways:

```
x = 12.21
x = .12 // one can omit the integer part when the decimal part is given
x = 12. // and vice versa
x = 12.5e10 // scientific notation
```

9.2.3 Strings

In Opa, text is represented by immutable utf8-encoded character strings. String literals follow roughly the common C/Java/-JavaScript syntax:

```
x = "hello!"
x = "\"" // special characters can be escaped with backslashes
```

Opa features `string insertions`, which is the ability to put arbitrary expressions in a string. This feature is comparable to string concatenations or manipulation of format strings, but is generally both faster, safer and less error-prone:

```
x = "1 + 2 is {1+2}" // expressions can be embedded into strings between curly braces
                     // evaluates to "1 + 2 is 3"
email(first_name,last_name,company) =
  "{String.lowercase(first_name)}.{String.lowercase(last_name)}@{company}.com"
my_email = email("Darth","Vader","deathstar") // evaluates to "darth.vader@deathstar.com"
```

More formally, the following characters are interpreted inside string literals:

characters	meaning
{	starts an expression (must be matched by a closing })
"	the end of the string
\\	a backslash character
\n	the newline character
\r	the carriage return character
\t	the horizontal tabulation character
\{	the opening curly brace
\}	the closing curly brace
\'	a single quote
\"	a double quote
\ anything else	forbidden escape sequence

9.3 Datastructures

9.3.1 Records

The only way to build datastructures in Opa is to use records. Since they are the only datastructure available, they are used pervasively and there is a number of syntactic shorthands available to write records concisely.

Here is how to build a record:

```
x = {} //  the empty record
x = {a=2; b=3} //  a record with the field "a" and "b"
x = {a=2; b=3;} //  you can add a trailing semicolon
x = {a=2 b=3} //  all the semicolons are optional
x = {`[weird-chars]` = "2"} //  a record with a field "[weird-chars]"

//  now various shorthands
x = {a} //  means {a=void}
x = {a b=2} //  means {a=void b=2}
x = {~a b=2} //  means {a=a b=2}
x = ~{a b} //  means {a=a b=b}
x = ~{a b c=4} //  means {a=a b=b c=4}
x = ~{a={b} c} //  means {a={b=void} c=c}, NOT {a={b=b} c=c}
```

The characters allowed in fields names are the same as the ones allowed in identifiers, which is described here.

You can also build record by *deriving* an existing record, i.e. creating a new record that is the same an existing record except for the given fields.

```
x = {a=1 b={c="mlk" d=3.}}
y = {x with a=3} //  means {a=3; b=x.b}
y = {x with a=3 b={e}} //  you can redefine as many fields as you want
                       //  at the same time (but not zero) and even all of them

//  You can also update fields deep in the record
y = {x with a.c = "po"} //  means {x with a = {x.a with c = "po"}}
                        //  whose value is {a=1 b={c="po" d=3.}}

//  the same syntactic shortcuts as above are available
y = {x with a} //  means {x with a=void}, even if it is not terribly useful
y = {x with ~a} //  means {x with a=a}
y = ~{x with a b={e}} //  means {x with a=a b={e}}
```

9.3.2 Tuples

Opa features syntactic support for pairs, triples, etc. — more generally *tuples*, ie, heteregenous containers of a fixed size.

```
x = (1,"mlk",{a}) //  a tuple of size 3
x = (1,"mlk") //  a tuple of size 2
x = (1,) //  a tuple of size 1
         //  note the trailing comma to differentiate a 1-uple
         //  from a parenthesized expression
         //  the trailing comma is allowed for any other tuple
         //  although it makes no difference whether you write it or not
         //  in these cases
//  NOT VALID: x = (), there is no tuple of size 0
```

Tuples are standard expressions: a N-tuple is just a record with fields `f1`, ..., `fN`. As such they can be manipulated and created like any record:

```
x = (1,"hello")
do @assert(x == {f1 = 1; f2 = "hello"})
do @assert(x.f1 == 1)
do @assert({x with f2 = "goodbye"} == (1,"goodbye"))
```

9.3.3 Lists

Opa also provides syntactic sugar for building *lists* (homogenous containers of variable length).

```
x = [] //  the empty list
x = [3,4,5] //  a three element list
y = [0,1,2|x] //  the list consisting of 0, 1 and 2 on top the list x
              //  ie [0,1,2,3,4,5]
```

Just like tuples, lists are standard datastructures with a prettier syntax, but you can build them without using the syntax if you wish. The same code as above without the sugar:

```
x : list = {nil}
x : list = {hd=3 tl={hd=4 tl={hd=5 tl={nil}}}}
x : list = {hd=0 tl={hd=1 tl={hd=2 tl=x}}}
```

9.4 Identifiers

In Opa, an identifier is a word matched by the following regular expression:
`([a-zA-Z_] [a-zA-Z0-9_]* | ` [^``\n\r] `)`
except the following keywords:
`_`, `as`, `do`, `else`, `if`, `match`, `then`, `type`, `with`.

In addition to these keywords, a few identifiers that can be used as regular identifiers in most situations but will be interpreted in some contexts:
`and`, `begin`, `css`, `db`, `end`, `external`, `forall`, `import`, `package`, `parser`, `rec`, `server`, `val`, `xml_parser`.
It is not advised to use these words as identifiers, nor as field names.

Any identifier may be written between backquotes: `x` and `` `x` `` are strictly equivalent. However, backquotes may also be used to manipulate any text as an identifier, even if it would otherwise be rejected, for instance because it contains white spaces, special characters or a keyword. Therefore, while `1\+2` or `match` are not identifiers, `` `1\+2` `` and `` `match` `` are.

9.5 Bindings

At toplevel, you can define an identifier with the following syntax:

```
one = 1
`hello` = "hello"
_z12 = 1+2
```

Tip
The compiler will warn you when you define a variable but never use it. The only exception is for variables whose name begins with _, in which case the compiler assumes that the variable is named only for documentation purposes.
As a consequence, you will also be warned if you use variables starting with _.
And for code generation, preprocessing or any use for which you don't want warnings, you can use variables starting with ___.

Of course, local identifiers can be defined too, and they are visible in the following expression:

```
two =
  one = 1 // an optional semicolon can be put after
  one + one

two =
  one = 1; // the exact same thing as above
           // can be used to make the code less ambiguous
  one + one

two =
  one = 1 // NOT VALID: syntax error because a local declaration
          // must be followed by an expression
```

9.6 Functions

9.6.1 Defining functions

In Opa, functions are regular values. As such, the follow the same naming rules as any other value. In addition, and a few syntactic shorcuts are available:

```
f(x,y) = // defining function f with the two parameters x and y
  x + y + 1

two =
  f(x) = x + 1 // functions call be defined locally, just like other values
  f(1)

f = x, y -> x + y + 1 // the exact same thing as above
                      // on the left of the arrow, you have comma separated parameters

// you can write functions in a currified way concisely:
f(x)(y) = x + y + 1
// or equivalently
f = x -> (y -> x + y + 1)
```

Caution
Note that there *must* be no space between the function name and its parameters, and no spaces between the function expression and its arguments.

```
f () = f() // NOT VALID: does not parse
x = f () // NOT VALID: parsed as
         // x = f
         // ()
```

9.6.2 Partial applications

From a function with N arguments, we may derive a function with less arguments by *partial application*:

```
add(x,y) = x+y
add1 = add(1,_) // which means add1(y) = add(1,y)
x = add1(2) // x is 3
```

Caution
Side effects of the arguments are computed at the site and time of partial application, not each time the function is called:

```
loop() = loop()
add1 = add(loop(), _) // this loops right now
                      // not when calling add1
```

All the underscores of a call are regrouped to form the parameters of a unique function in the same order are the corresponding underscores:

```
max3(x,y,z) = max(x,max(y,z))
positive_max = max3(0,_,_) // means positive_max(x,y) = max(0,x,y)
```

9.6.3 More definitions

We have already seen one way of defining anonymous functions, but there are two. The first way allows to functions of arbitrary arity:

```
add = x, y -> x + y
```

The second syntax allows to define only functions taking one argument, but it is more convenient in the frequent case when the first thing that your function does is match its parameter.

```
add1 =
| 0 -> 1
| 1 -> 2
| 2 -> 3
| _ -> error("Wow, that's outside my capabilities")
```

This last defines a function that does a pattern matching on its first argument (the meaning of this construct is described in Pattern-Matching).

```
add1(fresh) =
  match fresh with
  | 0 -> 1
  | 1 -> 2
  | 2 -> 3
  | _ -> error("Wow, that's outside my capabilities")
```

9.6.4 Operators

Since operators in Opa are standard functions, these two declarations are equivalent:

```
x = 1 + 2
x = `+`(1,2)
```

To be used as an infix operator, an identifier must contain only the following characters:

```
+ \ - ^ * / < > = @ | & !
```

Since operators as normal functions, you can define new ones:

```
`**` = Math.pow_i
x = 2**3 // x is 8
```

The priority and associativity of the operators is based on the leading characters of the operator. The following table show the associativity of the operators. Lines are ordered by the priority of operators, slower operators first.

leading characters	associativity
`| @`	left
`|| ?`	right
`&`	right
`= != > <`	left
`+ - ^`	left
`* /`	left

Caution
You cannot put white space as you wish around operators:

```
x = 1 - 2 // works because you have whitespace before and after the operator
x = 1-2 // works because you have no whitespace before and no white space after
x = 1 -2 // NOT VALID: parsed a unary minus
```

9.7 Type coercions

There are various reasons for wanting to put a type annotation on an expression:

- to document the code;
- to avoid value restriction errors;
- to make sure that an expression has a given type;
- to try to pinpoint a type error;
- to avoid anonymous cyclic types (by naming them).

The following demonstrates a type annotation:

```
x = [] : list(int)
```

Note that parameters of a type name may be omitted:

```
x = [] : list(list) // means list(list('a))
```

Type annotations can appear on any expression (but also on any pattern), and can also be put on bindings as shown in the following example:

```
x : list(int) = [] // same as s = [] : list(int)
f(x) : list(int) = [x] // annotation of the body of the function
                       // same as f(x) = [x] : list(int)
```

9.8 Grouping

Expressions can be grouped with parentheses or equivalently with `begin ... end`:

```
x = (1 + 2) * 3
x =
  if i_like_pascal then begin
    "begin end rocks"
  end else (
    "i like parens better"
  )
```

9.9 Modules

Functionalities are usually regrouped into modules.
The syntax resembles the one of record:

```
List = {{
  empty = []
  cons(hd,tl) = ~{hd tl}
}}
```

By opposition to records, modules do not offer any of the syntactic shorthands: no `~{{x}}`, no `{{x}}`, nor any form of module derivation: no `{{List with cons(hd,tl) = [hd,hd|tl]}}`.
On the other hand, the content of a modules are *not* field definitions, but *bindings*. This means that the fields of a module can access the other fields:

```
m = {{
  x = 1
  y = x // x is in scope
}}

r = {
  x = 1
  y = x // NOT VALID: x is unbound
}
```

Note that, by opposition to the toplevel, modules contain *only* bindings, no type definitions.

The bindings of a module are all mutually recursive (but still subject to the recursion check, once the recursivity has been reduced to the strict necessary):

```
m = {{
  x = y
  y = 1
}}
```

This will work just fine, because this is reordered into:

```
m = {{
  y = 1
  x = y
}}
```

where you have no more recursion.

Caution
Since the content of a module is recursive, it is not guaranteed that the content of a module is executed in the order of the source.

9.10 Sequencing computations

In Opa the toplevel is executed, and so you can have expressions at the toplevel:

```
do println("Executed!")
```

The `do` construct can be used to compute an expression and discard its result.

```
x =
  do println("Dibbs!") // cleaner than saying _unused_name = println("Dibbs!")
                       // but equivalent (almost, see the warning section)
  do println("Aww...")
  1
```

Note that the definitions of identifiers and the `do` construct are always followed by an expression in expressions, but never at the toplevel.

9.11 Datastructures manipulation and flow control

The most basic way of deconstructing a record is to *dot* (or "*dereference*") the content of an existing field.

```
x = {a=1 b=2}
do @assert(x.a == 1)
c = x.c // NOT VALID: type error, because x does not have a field c
```

Note that the dot is defined only on records, not sums. For sums, something more powerful is needed:

```
x = {true} : bool
do @assert(x.true) // NOT VALID: type error
```

To deconstruct both records and sums, Opa offers *pattern-matching*. The general syntax is:

```
do
  match <expr> with
  | <pattern_1> -> <expr_1>
  | <pattern_2> -> <expr_2>
  ...
  | <pattern_n> -> <expr_n>
  end
```

Note that keywords `with` and `end` are optional, as well as `|`. For clarity, it is however generally a good idea to restrain from omitting them.

When evaluating this extract, the result of `<expr>` is compared with `<pattern_1>`. If both *match*, i.e. the have the same shape, `<expr_1>` is executed. Otherwise, the same result is compared with `<pattern_2>`, etc. If no pattern matches, then an error happens and the execution flow stops.

The specific case of pattern matching on boolean can be abreviated using a standard `if-then-else` construct:

```
do if 1 == 2 then
     println("Who would have known that 1 == 2?")
   else
     println("That's what I thought!")

// if the else branch is omitted, it default to void
do if 1 == 2 then
     println("Who would have known that 1 == 2?")

// or equivalently
do match 1 == 2 with
   | {true} -> println("Who would have known that 1 == 2?")
   | {false} -> println("That's what I thought!")
   end
```

Caution

Since the `match` construct can end with `end`, beware of its interaction with `begin`:

```
do
  begin
    match 0 with
    | _ -> void
  end
```

This is a syntax error because the `end` is actually part of the `match` construct, and so the `begin` has no corresponding `end`.

Tip

The same way that `f(x,_)` means (roughly) `y -> f(x,y)`, `_.label` is a shorthand for `x -> x.label`, which is convenient when combined with higher order:

```
l = [(1,2,3),(4,5,6)]
l2 = List.map(_.f3,l) // extract the third elements of the tuples of l
                      // ie [3,6]
```

9.11.1 Patterns

Generally, patterns appear as part of a `match ... with` construct. However, they may also be used at any place where you bind identifiers.

Syntactically, patterns look like a very limited subset of expressions:

```
1 // an integer pattern
-2.3 // a floating point pattern
"hi" // a string pattern, no embedded expression allowed
{a=1 ~b} // a (closed) record pattern, equivalent to {a=1 b=b}
[1,2,3] // a list pattern
(1,"p") // a tuple pattern
x // a variable pattern
```

On top of these constructions, you have

```
{a=1 ...} // open record pattern
_ // the catch all pattern
<pattern> as x // the alias pattern
{~a=<pattern>} // a shorthand for {a=<pattern> as a}
```

```
<pattern> | <pattern> // the 'or' pattern
                      // the two sub patterns must bind the same set of identifiers
```

When the expression `match <expr> with <pattern> -> <expr2> | ...` is executed, `<expr>` is evaluated to a value, which is then matched against each pattern in order until a match is found.

9.11.2 Matching rules

The rules of pattern-matching are simple:

- any value matches pattern _;
- any value matches the variable pattern `x`, and the value is *bound* to identifier `x`;
- an integer/float/string matches an integer/float/string pattern when they are equal;
- a record (including tuples and lists) matches a closed record pattern when both record have the same fields and the value of the fields matches the pattern component-wise;
- a record (including tuples and lists) matches an open record pattern when the value has all the fields of the pattern (but can have more) and the value of the common fields matches the pattern component-wise;
- a value matches a `pat as x` pattern when the value matches `pat`, and additionally it binds `x` to the value;
- a value matches a `or` pattern is one of the value matches one of the two sub patterns;
- in all other cases, the matching fails.

Pattern-matching does not test for equality
Consider the following extract:

```
x = 1
y = 2
do match y with
  | x -> println("Hey, 1=2")
  | _ -> println("Or not")
```

You may expect this code to print result "Or not". This is, however, not what happens. As mentioned in the definition of matching rules, pattern `x` matches *any value* and binds the result to identifier `x`. In other words, this extract is equivalent to

```
x = 1
y = 2
do match y with
  | z -> println("Hey, 1=2")
  | _ -> println("Or not")
```

If executed, this would therefore print "Hey, 1=2". Note that, in this case, the compiler will reject the program because it notices that the two patterns test for the same case, which is clearly an error.

A few examples:

```
list_is_empty(l) =
  match l with
  | [] -> true
  | [_|_] -> false

// and without the syntactic sugar for lists
// a list is either {nil} or {hd tl}
head(l) =
  match l : list with
  | {nil} -> @fail
  | ~{hd ...} -> hd
```

Warning
At the time of this writing, support for `or` patterns is only partial. It can only be used at the toplevel of the patterns, and it duplicates the expression on the right hand side.

Warning
At the time of this writing, support for `as` patterns is only partial. In particular, it cannot be put around open records, although this should be available soon.

Warning
A pattern cannot contain an expression:

```
is_zero(x) = // works fine
match x with
| 0 -> true
| _ -> false

// wrong example
zero = 0
is_zero(x)
match x with
| zero -> true
| _ -> false
// does not work because the pattern defines zero
// it does not check that the x is equal to zero
```

Caution
You cannot put the same variable several times in the same pattern:

```
on_the_diagonal(position) =
  match position with
  | {x=value y=value} -> true
  | _ -> false
// this is not valid because you are trying to give the name value
// to two values

// this must be written
on_the_diagonal(position) =
  position.x == position.y
```

9.12 Loops

At this stage, you may wonder about how to write loops, iterators, etc. in Opa.

Surprisingly, Opa does not offer a specific syntax for loops. Rather, Opa offers *function loops* as part of the standard library.

```
// printing a chain 10 times
// repeat has type : int, (-> void) -> void
do repeat(10,(-> println("Hello!")))

// printing the integer for 1 to 10
```

```
// inrange has type int, int, (int -> void) -> void
do inrange(1,10,(i -> println("{i}")))

// summing integer starting from 1 until the sum is greater than 50
// while has type: 'state, ('state -> ('state, bool)) -> 'state
~{sum ...} =  // we only return the sum, ie the first field of the pair
   while({sum=0 i=1},
         (~{sum i} ->
            sum = sum + i
            i = i + 1
            ~{sum i}, (sum <= 50)))

// the same function with the for function
// for has type: 'state, ('state -> 'state), ('state -> bool) -> 'state
~{sum ...} =
  for(
    {sum=0 i=1}, // the initial state
    (~{sum i} -> {sum=sum+i i=i+1}), // the function that computes the next state
    (~{sum ...} -> sum <= 50) // the function that tells if we should continue
  )

/* the equivalent with an imperative syntax:
sum = 0

for (i = 1; sum <= 50; i=i+1) {
    sum=sum+i
}
*/
```

In the above,

- assignments `sum=0; i=1` correspond to the record `{sum=0 i=1}` above;;
- the body of the loop `sum=sum+i; i=i+1` corresponds to the function `~{sum i} -> {sum=sum+i; i=i+1}`;
- the loop condition `sum ⇐ 50` corresponds to `~{sum ...} -> sum ⇐ 50`.

Additional loop functions may be easily created, either by building them from these functions, or through Section 9.14.

9.13 Parser

Opa features a builtin syntax for building text parsers, which are first class values just as functions. The parsers implement *parsing expression grammars*, which may look like regular expressions at first but do *not* behave anything like them.

An instance of a parser:

```
sign_of_integer =
  parser "-" [0-9]+ -> {negative}
       | [0-9]+ -> {positive}
```

A parser is composed of a disjunction of `<list-of-subrules> (-> <semantic-action>)?`, separated by a `|`. When the semantic action is omitted, it defaults to the `text` that was parsed by the left hand side.

A subrule consists of:

1. an optional binder that names the result of the subrule. it can be:
    a. `x=` to name the result `x`

   b. `~` only when followed by a long identifier. In the case, the name bound is the last component. For instance, `~M.x*` means `x=M.x*`

2. an optional prefix modifier (`!` or `&`) that lookahead for the following subrule in the input
3. a basic subrule
4. an optional suffix modifier (`?`, `*`, `+`), that alters the basic in the usual way

And the basic subrule is one of:

```
"hello {if happy then ":)" else ":("}" // any string, including strings
                                       // with embedding expressions
'hey, I can put double quotes in here: ""' // a string inside single quotes
                                           // (which cannot contain embedded expressions)
Parser.whitespace // a very limited subset of expression can be written inline
                  // only long identifiers are allowed
                  // the expression must have the type Parser.general_parser
{Parser.whitespace} // in curly braces, an arbitrary expression
                    // with type Parser.general_parser
. // matches a (utf8) character
[0-9a-z] // character ranges
         // the negation does not exist
[\-\[\]] // in characters ranges, the minus and the square brackets
         // can be escaped
( <parser_expression> ) // an anonymous parser
```

Caution
Putting parentheses around a parser can change the type of the parenthesized parsers:

```
parser x=.* -> ... // x as type list(Unicode.character)
parser x=(.*) -> ... // x has type text
```

This is because the default value of a parenthesized expression is the text parsed.
This is the only way of getting the text that was matched by a subrule.

A way to use a parser (like `sign_of_integer`) to parse a string is to write:

```
Parser.try_parser(sign_of_integer,"36")
```

For an explanation of how parsing expression grammars work, see http://en.wikipedia.org/wiki/Parsing_expression_grammar.
Here is an example to convince you that even if it looks like a regular expression, you can not use them as such:

```
parser "a"* "a" -> void
```

The previous parser will *always* fail, because the star is a greedy operator in the sense that it matches the longest sequence possible (by opposition with the longest sequence that makes the whole regular expression succeed, if any):
`"a"*` will consume all the `"a"` in the strings, leaving none for the following `"a"`.

9.14 Recursion

By default, toplevel functions and modules are implicitely recursive at toplevel, while local values (including values defined in functions) are not.

```
f() = f() // an infinite loop

x =
  f() = f() // NOT VALID: f is unbound
```

```
  void

x =
  rec f() = f() // now f is visible in its body
  void

f() = g() // mutual recursion works without having
g() = f() // to say 'rec' anywhere at toplevel

x =
  rec f() = g() // local mutually recursive functions must
  and g() = f() // be defined together with a 'rec' 'and'
                // construct
  void
```

Recursion is only permitted between functions, although you can have recursive modules if it amounts to valid recursion between the fields of the module:

```
m = {{
  f() = m2.f()
}}
m2 = {{
  f() = m.f()
}}
```

This is valid, because it amounts to:

```
rec m_f() = m2_f()
and m2_f() = m_f()
m = {{ f = m_f }}
m2 = {{ f = m2_f }}
```

Which is a valid recursion.

Opa also allows arbitrary recursion (in which case, the validity of the recursion is checked at runtime), but it must be indicated explicitely that is what is wished for with `val`:

```
rec val sess = Session.make(callback)
and callback() = do something with sess
```

Tip
The compiler will not allow you to put `val` when not necessary (i.e. on functions).

Please note that the word `val` is meant to define *recursive* values, but not meant to define *cyclic* values:

```
rec val x = [x]
```

This definition is invalid, and will be rejected (statically in this case despite the presence of the `val` because it is sure to fail at runtime).

Of course, most invalid definitions will be only detected at runtime:

```
rec val x = if true then x else 0
```

9.15 Directives

Many special behaviours appear syntactically as directives, starting with a `@`. A directive can impose arbitrary restrictions on its arguments.
They are usually used because we want to make it clear in the syntax that something special is happening, that we do not have a regular function call.

Some directives are expressions, while some directives are annotations on bindings, and they do not appear in the same place.

```
do if true then void else @fail // @fail appears only in expressions
@expand `=>`(x,y) = not(x) || y // the lazy implication
                                // @expand appears only on bindings
                                // and precedes them
```

Here is a full list of (user-available) expression directives, with the restriction on them:

@assert
: Takes one boolean argument
Raises an error when its argument is false. The code is removed at compile time when the option --no-assert is used.

@fail
: Takes an optional string argument
Raises an error when executing (and show the string if any was given) Meant to be used when something cannot happen

@todo
: Takes no argument
Behaves like `@fail` except that a warning is shown at each place when this directive happens (so that you can conveniently replace them all with actual code later)

@toplevel
: Takes no argument, and must be followed by a field access
`@toplevel.x` allows to talk about the `x` defined at toplevel, and not the `x` in the local scope

@unsafe_cast
: Takes one expression
This directive is meant to bypass the typer. It behaves as the identity of type `'a -> 'b`.

Here is a full list of (user-available) bindings directives, with the restriction on them:

@comparator
: Takes a typename
Overrides the generic comparison for the given type with the function annotated.

@deprecated
: Takes one argument of the following kind: `{hint = string literal} / {use = string literal}`
Generates a warning to direct users of the annotated name. The argument is used when displaying the warning (at compile time).

@expand
: Takes no argument, and appears only on toplevel functions.
The directive calls to this function will be macro expanded (but without name clashes). This is how the lazy behaviour of `&&`, `\|\|` and `?` is implemented.

@private
: Takes no argument, only on toplevel binding or module fields.
The identifier tagged is not visible outside the module (or package if at toplevel).

@public
: Takes no argument, only on toplevel binding or module fields.
The identifier tagged is visible outside the current module/package (this is the default).

@stringifier
Takes a typename
Overrides the generic stringification for the given type with the function annotated:

```
@stringifier(bool) to_string(b: bool) = if b then "true" else "false"
```

9.16 Foreign function interface

Foreign functions, or *system bindings*, are standard expressions. To use one, simply write the key (see the corresponding chapter) of your binding between `%%`:

```
x = (%% BslPervasives.int_of_string %%)("12") // x is 12
```

9.17 Separate compilation

At the toplevel only, you can specify information for the separate compilation:

```
package myapp.core // the name of the current package
import somelib.* // which package the current package depends on
```

Inside the import statement, you can have shell-style brace and glob expansion:

```
import graph.{traversal,components}, somelib.*
```

Tip
The compiler will warn you whenever you import a non existing package, or if one of the alternatives of a brace expansion matches nothing, or a if a glob expansion matches nothing.

Beware that the toplevel is common to all packages.
As a consequence, it is advised to define packages that export only modules, without other toplevel values.

9.18 Type expressions

Type expressions are used in type annotations, and in type definitions.

9.18.1 Basic types

The three data types of Opa are written `int`, `float` and `string`, like regular typenames (except that these names are actually not valid typenames).
Typenames can contain dots without needing to backquote them: `Character.unicode` is a regular typename.

9.18.2 Record types

The syntax for record type works the same as it does in expressions and in patterns:

```
x : {useless} = @fail // means {useless:void}
x : ~{a b} = @fail // means {a:a b:b}, where a and b are typenames
x : ~{list} = @fail // means the same as {list:list}
                    // this is valid in coercions because you can omit
                    // the parameters of a typename (but not in type definitions)
x : ~{int} = @fail // means {int:'int'}, so you must have defined a type 'int'
                   // for this definition to be valid
x : {a b ...} = {a b c} // you can give only a part of the fields in type annotations
```

9.18.3 Tuple types

The type of a tuple actually looks like the tuple:

```
(a,b) : (int,float) = (1,3.4)
```

9.18.4 List types

There is no syntax for the type of list, you simply use `list`.

```
x : [int] = [1] // NOT VALID
x : list(int) = [1]
```

9.18.5 Sum types

Now, record expressions do not have records type (in general), they have sum types, which are simply unions of record types:

```
x : {true} / {false} = {true}
x : {true} / ... = {true} // sum types can be partially specified, just like record types
```

9.18.6 Type names

Types can be given names, and of course you can refer to names in expressions:

```
x : list(int) = [1] // the parameters of a type are written just like a function call
x : bool = 1 // except that when there is no parameter, you don't write empty parentheses
x : list = [1] // and except that you can omit all the parameters of a typename altogether
               // (which means 'please fill up with fresh variables for me')
```

9.18.7 Variables

Variables begin with an apostrophe except _:

```
x : list('a) = []
x : list(_) = [] // _ is an anonymous variable
```

9.18.8 Function types

Function types are written the same way as anonymous functions:

```
max3 : int, int, int -> int = x, y, z -> max(x, max(y, z))
```

9.19 Type definitions

A type definition allows to give a name to a type.

It simply consists of an identifier, a list of parameters, a set of directives and a body.
Since type definitions can only appear at the toplevel, and the toplevel is implicitely recursive, all the type definitions of a package are mutually recursive.

Here are the most common types in opa as defined in the standard library:

```
type void = {} // naming a record
type bool = {true} / {false} // naming a type sum
type option('a) = {none} / {some:'a} // a parameterized definition of a type sum
type list('a) = {nil} / {hd:'a tl:list('a)} // a recursive and parameterized
                                            // definition of a type sum
```

In addition to type expressions, the body of a type definition can be an external types, ie types that represent foreign objects, used when interfacing with other languages.

```
type continuation('a) = external
```

9.19.1 Type directives

There are currently only two directives that can be put on type definitions, and they both control the visibility of the type.

The first one is `@abstract`, which hides the implementation of a type to the users of a library:

```
package test1
@abstract type Test1.t = int
Test1 = {{
  x : t = 1
}}
```

Abstracting forces the users to go through the interface of the library to build and manipulate values of that type.

```
package test2
import test1
x = Test1.x + 1 // this is a type error, since in the package test2
                // the type Test1.t is not unifiable with int anymore
```

The second directive is `@private`, which is a type that is not visible from the outside of the module (not even its name). When a type is private, values with that type cannot be exported

```
package test1
@private type t = int
Test1 = {{
  x : t = 1 // will not compile since the module exports
            // Test1.x that has the private type t
}}
```

9.20 Formal description

This syntax recapitulates the syntactic constructs of the language.

9.20.1 Conventions

The following conventions are adopted to describe the grammar. The following defines `program` with the production `prod`.

```
program ::= prod
```

A reference to the rule `program` between parens:

```
( <program> )
```

A possibly empty list of `<rule>` separated by `<sep>` is written:

```
<rule>* sep <sep>
```

A non empty list of `<rule>` separated by `<sep>` is written:

```
<rule>+ sep <sep>
```

9.20.2 The opa language

A source file is a `<program>`, defined as follows:

```
program ::= <declaration>* sep ;?
declaration ::=
  | <type-definition>
  | <binding>
  | <do>
  | <package-declaration>
  | <package-import>
```

The rules related to separate compilation:

```
package-declaration ::=
  | package <package-ident>
package-import ::=
  | import <package-expression>* sep ,
package-expression ::=
  | { <package-expression>* sep , }
  | <package-expression> *
  | <package-ident>
package-ident ::= [a-zA-Z0-9_.\-]
```

Some rules used in many places:

```
field ::= <ident>

literal ::=
  | <integer-literal>
  | <float-literal>
  | <string-literal>

long-ident ::=
  | <long-ident> . <ident>
  | <ident>
```

The syntax of the types:

```
type-definition ::=
  | <type-directive>* type <type-bindings>
type-bindings ::=
  | <type-binding>* sep and
type-binding ::=
  | <type-ident> ( <type-var>* sep "," ) = <type-def>
type-def ::=
  | <type>
  | external
type ::=
  | <type>* sep , -> <type>
  | <tuple-type>
  | <record-type>
  | <type-ident> ( <type>* sep , )
  | int
  | string
  | float
  | <type> / <type>+ sep / <sum-type-end>?
  | <type> <sum-type-end>
  | _
  | <type-var>
  | forall ( <type-var>* sep , ) . <type>
  | ( <type> )

<tuple-type> ::=
  | ( <type> , )
  | ( <type> , <type>+ sep , ,? )

record-type ::=
  | ~? { <record-type-field>* sep ;? <record-type-end>? }
record-type-field ::=
  | ~ <field>
  | <field> : <type>
record-type-end ::=
  | ...
  | <row-var>

sum-type-end ::=
  | ...
  | <col-var>
type-ident ::= <long-ident>
```

The syntax of the patterns:

```
pattern ::=
  | <literal>
  | <record-pattern>
  | <tuple-pattern>
  | <list-pattern>
  | <pattern> as <ident>
  | <pattern> | <pattern>
  | <pattern> : <type>
  | ( <pattern> )
  | _

<tuple-pattern> ::=
  | ( <pattern> , )
  | ( <pattern> , <pattern>+ sep , ,? )

<record-pattern> ::=
  | ~? { ...? }
```

```
  | ~? { <record-pattern-field>+ sep ;? ;? ...? }
<record-pattern-field> ::=
  | ~? <field> <coerce>?
  | ~? <field> <coerce>? = <pattern>

<list-pattern> ::=
 | [ pattern+ sep , | pattern ]
 | [ pattern* sep , ]
```

The syntax of the bindings:

```
binding ::=
  | rec <rec-binding>+ sep and
  | <non-rec-binding>

non-rec-binding ::=
  | <binding-directive> <ident-binding>
  | <binding-directive> <val-binding>

rec-binding ::=
  | <ident-binding>
  | val <val-binding>

ident-binding ::=
  | <ident> <params>+ <coerce>? = <expr>
  | <ident> <coerce>? = <expr>

params ::=
  | ( <pattern>* sep "," )

val-binding ::=
  | <pattern> = <expr>

do ::=
  | do <expr>

coerce ::=
  | : <type>
```

The syntax of the expressions, except the parsers:

```
expr ::=
  | <do> ;? <expr>
  | <binding> ;? <expr>
  | match <pattern> with? |? <rule>* sep |? end?
  | if <expr> then <expr> else <expr>
  | if <expr> then <expr>
  | <record>
  | <module>
  | <tuple>
  | <list>
  | <grouping>
  | <expr> : <type>
  | <literal>
  | <expr-or-underscore> ( <expr-or-underscore>* sep "," )
  | <expr-or-underscore> . field
  | <directive>
  | <lambda>
  | <expr-or-underscore> <op> <expr-or-underscore>
  | - <expr>
  | <sysbinding>
  | <ident>
```

```
  | <parser>

lambda ::=
  | <pattern>* sep , -> <expr>
  | | <rule>*

rule ::=
  | <pattern> -> <expr>

grouping ::=
  | ( <expr> )
  | begin <expr> end

record ::=
  | ~? { <record-field>* sep ;? }
  | ~? { <expr> with <record-field>+ sep ;? }
record-field ::=
  | ~? <field> <coerce>?
  | <field> <coerce>? = <expr>

tuple ::=
  | ( <expr> , )
  | ( <expr>, <expr>+ sep , ,? )

list ::=
  | [ <expr>+ sep , | <expr> ]
  | [ <expr>* sep , ]

module ::=
  | {{ <non-rec-binding>+ sep ;? }}

expr-or-underscore ::=
  | <expr>
  | _
```

The syntax of the parsers:

```
parser ::=
  | parser <parser-rules>
parser-rules ::=
  | <parser-rule>+ sep |
parser-rule ::=
  | <parser-prod>+ -> <expr>
  | <parser-prod>+
parser-prod ::=
  | <parser-name>? <subrule-prefix>? <subrule> <subrule-suffix>?
parser-name ::=
  | <ident> =
  | ~
subrule-prefix ::=
  | &
  | !
subrule-suffix ::=
  | *
  | +
  | ?
subrule ::=
  | <character-set>
  | .
  | <string-literal>
  | { <expr> }
  | <long-ident>
```

Chapter 10

Developing for the web

In this chapter, we recapitulate the web-specific constructions of the Opa language, including management of XML and XHTML, CSS, but also client-server security.

10.1 Syntax extensions

The syntax of expression is extended by the following rules:

```
expr ::=
| <xhtml>
| <ip>
| <id>
| css <css>
| css { <style-expr> }
| <action-list>
| <xml-parser>

action-list ::=
| [ <action>* sep , ]

action ::=
| <action-selector> <action-verb> <expr>

action-selector ::=
| <action-selector-head> <action-selector-property>?

action-selector-head ::=
| . <ident>
| . { <expr> }
| <id>

action-selector-property ::=
| -> <ident>
| -> { <expr> }

ip ::=
| <byte> . <byte> . <byte> . <byte>

id ::=
| # <ident>
| # { <expr> }

xhtml ::=
| <> <xhtml-content>* </>
```

```
| < <xhtml-tag> <xhtml-attribute>* />
| < <xhtml-tag> <xhtml-attribute>* > <xhtml-content>* </ <xhtml-tag>? >

xhtml-content ::=
| <xhtml>
| <xhtml-text>
| { <expr> }

xhtml-tag ::=
| <xhtml-name> : <xhtml-name>
| <xhtml-name>

xhtml-attribute ::=
| style = <xhtml-style>
| class = <xhtml-class>
| on{click,hover,...} =
| options:on{click,hover,...} =
| <xhtml-tag> = <xhtml-attr-value>

xhtml-attr-value ::=
| <string-literal>
| <single-quote-string-literal>
| { <expr> }
| <id>

xhtml-style ::=
| " <style-expr> "
| { <expr> }

xhtml-class ::=
| ' <xhtml-name>* '
| " <xhtml-name>* "
| <xhtml-name>
| { <expr> }

style-expr ::=
| ...

css ::=
| ...

xml-parser ::=
| xml_parser <xml-parser-rules>

xml-parser-rules ::=
| |? <xml-parser-rule>* sep | end?

xml-parser-rule ::=
| <xml-named-pattern>+ -> <expr>

xml-named-pattern ::=
| <ident> <xml-parser-suffix>?
| <ident> = <xml-pattern> <xml-parser-suffix>?
| parser? <parser-prod>+

xml-pattern ::=
| < <xhtml-tag> <xml-pattern-attribute>* />
| < <xhtml-tag> <xml-pattern-attribute>* > <xml-named-pattern>* </ <xhtml-tag> >
| _
| { <expr> }
| ( <xml-parser-rules> )
```

```
xml-pattern-attribute ::=
| <xhtml-tag> <xml-pattern-attribute-rhs>?

xml-pattern-attribute-rhs ::=
| <xml-pattern-attribute-value>
| ( <xml-pattern-attribute-value> as <ident> )

xml-pattern-attribute-value ::=
| <string-literal>
| { <expr> }
| _

xml-parser-suffix ::=
| ?
| +
| *
| { <expr> }
| { <expr> , <expr> }
```

Additionaly, some more directives are available.

And finally, the identifiers `server` and `css` have a special role at toplevel.

10.1.1 Xhtml, xml

Although xhtml and xml is not a built-in datastructure, there is a shorthand syntax for building it.

```
div = <div class="something">Hey</div>
```

which is a shorthand for the following structure:

```
{ namespace =  Xhtml.ns_uri
  tag = "div"
  args = []
  specific_attributes = {
    some = {
      class = ["something"] ;
      style = [] ;
      events = [] ;
      events_options = [] ;
      href = {none}
    }
  }
  content = [{ text = "Hey" }]
} : xhtml
```

The syntax only allows to build xhtml (not html, so there is no implicit closing of tags). On the other end, it is allowed to close any tag with the empty tag:

```
div = <div class="something">Hey</>
```

You can build fragments of xhtml with an empty tag:

```
<> First piece <div class="something">Hey</div> Third piece </>
```

Just like you can insert expressions into strings, you can insert expressions into xhtml. Here is the previous examples rewritten with insertions:

```
string1 = "First piece"
div = <div class={["something"]}>Hey</div>
string2 = "Third piece"
<> {string1} {div} {string2} </>
```

Note
You can only insert expressions in the content of a tag or the value of expressions, you cannot insert expressions instead of a tag name for instance.

```
tag = "div"
some_xhtml = <{tag}>Hello</> // NOT_VALID
```

In xhtml literals, the usual comments `/* */` and `//` do not work anymore. Instead, you have xhtml comments `<!-- -->`, which really are comments and not a structure representing comments (so these comments will never appear at runtime, you cannot send them to the client etc.).

There are a few attributes that are dealt with specially in xhtml literals.

Style The value associated to the style attribute should be of type `Css.properties`. For convenience, it can be written as a string, just like you would in html files, but it is *not* a string and it *will* be parsed.

Class The value associated to the class attribute has a similar behaviour to the one of style. Its value has the type `list(string)` but it can be written as a string, that is going to be parsed.

Onclick,ondblclick,onmouseup,... The full list is actually the type of Dom.even.kind. The value of this attribute is a `FunAction.t`, ie `Dom.event -> void`.

Options:onclick, options:ondblclick, ... The value of this attribute is a `list(Dom.event_option)`.

10.1.2 Namespaces

By default, when you use the syntax for xml literals, you actually build xhtml (the default namespace is the one of xhtml, some attributes have special meaning, etc.). This can be disabled by putting a @xml around an xhtml literal.

```
_ = @xml(<example attr="value"/>)
```

You can build xml with any namespace, not necessarily only the empty one or the one of xhtml.

```
some_soap = @xml(
 <soap:Envelope
     xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
     soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
   <soap:Body xmlns:m="http://www.example.org/stock">
     <m:hello>Hello world</m:hello>
   </soap:Body>
 </soap:Envelope>
)
```

When you define a namespace with a xmlns attribute, the namespace is defined in the current litteral only. If you insert a piece of html, it must also defines the namespace:

```
body = @xml(
   <soap:Body xmlns:m="http://www.example.org/stock">
     <m:hello>Hello world</m:hello>
   </soap:Body>
) // this is not valid because the namespace soap is not in scope
some_soap = @xml(
 <soap:Envelope
     xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
     soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
   {body}
 </soap:Envelope>
)
```

To solve this problem and to factorize the namespaces, you can name them with a normal binding:

```
`xmlns:soap`="http://www.w3.org/2001/12/soap-envelope"
body = @xml(
   <soap:Body xmlns:m="http://www.example.org/stock">
     <m:hello>Hello world</m:hello>
   </soap:Body>
) // this not valid because `xmlns:soap` is in scope
some_soap = @xml(
 <soap:Envelope
     soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
   {body}
 </soap:Envelope>
)
```

10.1.3 Parser

Xml is often used as an intermediate structure, and as such, it should be convenient to transform it into a less generic structure.

Opa features a special pattern matching like constructs that is meant for this:

```
my_parser = xml_parser
    | <x>parser valx=Rule.integer</><y>parser valy=Rule.integer</> -> {x=valx y=valy}
```

It defines a parser that transforms xml such as

```
<x>12</x><y>13</y>
```

into

```
{x=12 y=13}
```

This syntax resembles the one of pattern matching, and it acts the same:
The xml begin parsed is matched against the patterns in order until a match is found.
The right hand side of the pattern is then executed.

We detail here how `xml_parser` patterns behave:

- One rule is composed of a list of patterns.
  Any xml matches the empty list of patterns. Note that the xml doesn't have to be empty.

  An xml matches a non empty list of patterns if the xml matches the head pattern and its siblings (if it is a fragment, or the empty fragment otherwise) match the remaining patterns.
- An xml matches the pattern `parser <parser-prod>+` if its first node is a text node that is accepted by the parser.
  The bindings done in the parser are in the scope of the action.
- An xml matches the pattern `x=<subrule>` when the xml matches the <subrule> and the result of <subrule> is then bound to x.
- An xml matches the pattern `x<suffix>` when the xml matches the pattern `x=_<suffix>`
- An xml always matches the subrule `<subrule>*`.
  As many nodes as possible are matched, and the list of results returned by those matched is returned.
- An xml matches the subrule `<subrule>+` when the xml matches <subrule> at least once.
  As many nodes as possible are matched, and the list of results returned by those matched is returned.
- An xml always matches the subrule `<subrule>?`.
  A node is matched if possible, and its result wrapped into an option is returned.
- An xml matches the subrule `<subrule>{min,max}` when the xml matches <subrule> at least `min` times. As many nodes as possible (but stopping at `max`) are matched, and the list the results of these matches is returned.

- An xml matches the subrule `<subrule>{number}` when it matches `<subrule>{number,number}`.
- An xml matches the subrule _ if it contains at least a node. This node is then returned.
- An xml matches the subrule `{<expr>}` when the calling the xml_parser **<expr>** accepts the xml. In that case, the result of the xml_parser is returned.
- An xml matches the subrule `<tag attributes>content</>` when the xml begins with a tag node and
  - the tag of the node match the tag of the pattern
  - the attributes of the node match the attributes of the pattern
  - the content of the node match the content of the pattern
- A tag matches a tag of a pattern when they have the same namespace and the same name.
- A list of attributes matches a list of attributes pattern when each attribute
  - has no counterpart in the list of attribute pattern
  - or has a counterpart in the attribute pattern list and their value match
    (a counterpart being an attribute with the same namespace and name)
- An attribute value always matches _
- An attribute value always matches the absence of value pattern (`<tag attr=1>` matches the pattern `<tag attr/>`) The value of the attribute is bound to name of the attribute.
- An attribute value matches `{ <expr> }` when the parser `<expr>` accepts the value. The return value is bound to the name of the attribute.
- An attribute value matches `<string-literal>` when it matches `{ Parser.of_string(<string-literal>) }` The return value is bound to the name of the attribute.
- An attribute value matches `(<value> as <ident>)` when it matches **<value>**.
  In that case, the value of the matching is bound to `<ident>` instead of being bound to the name of the attribute.

Important
All whitespace only text nodes are discarded during parsing.

```
xml = @xml(<>"  "</>)
p = xml_parser
    | parser .* -> {}
```

`p` would fail at parsing `xml`, because it behaves exactly as if `xml` were defined as

```
xml = @xml(<></>)
```

10.1.4 Ip address

You can write constant ip addresses with the usual syntax:

```
127.0.0.1
```

This expression has type `IPv4.ip`.

10.1.5 Directives

The set of expression directives is extended by the following directives:

@sliced_expr
Takes one static record with the field `server` and `client` (containing arbitrary expressions)
Meaning described in the client-server distribution.

@static_resource , @static_resource_directory

@xml
Takes an xml literal
The default namespace in the literal in the empty uri, not the xhtml uri.

The set of binding directives is also extended:

@both , @both_implem , @client , @publish , @publish_async
Takes no argument, appear at the toplevel or inside toplevel modules
Meaning described in the client-server distribution.

@serializer
Takes a typename
Overrides the generic serialization and deserialization for the given type with the pair of functions annotated.

@server , @server_private
Takes no argument, appear at the toplevel or inside toplevel modules
Meaning described in the client-server distribution.

@xmlizer
Takes a typename
Overrides the generic transformtion to xml for the given type with the function annotated.

10.1.6 Css

Opa allows you to define css (as a datastructure) using the syntax of css:

```
mycss = css
body {
  background: white none repeat scroll top left;
  color: #4D4D4D;
}
```

Now this is just a datastruture that you can manipulate like any other. To actually serve it to the clients, you need to register it, which is done by defining a variable with name `css` at toplevel.

```
css = [my_css]
```

The right hand side should be of type `list(Css.declaration)`.

One exception is that it is allowed to say simply:

```
css = css
body {
  background: white none repeat scroll top left;
  color: #4D4D4D;
}
```

The right hand side of css is of type `Css.declaration`, but in the special case when it is a literal, it gets automatically promoted to a list of one element.
It allows for a slightly lighter syntax when the css of your application is defined in one block.

To define css in opa, you simply declare a variable with name `css` at toplevel.

The style attribute of xhtml constructs is also parsed specially. Its content looks like a string but is actually a structure.

```
<div style="top: 0px; left: 29px; position: absolute; ">
```

This structure can be built in expression, you do not need a style attribute to build it:

```
css { top: 0px; left: 29px; position: absolute; }
```

The previous div was simply a shorthand for:

```
<div style={ css { top: 0px; left: 29px; position: absolute; } }>
```

10.1.7 Defining servers

The way to define a server in Opa is to define a toplevel value named `server`, whose right hand side should be an expression of type `service`.

```
server = one_page_server("Hello", -> <h1>Hello World</h1>)
```

10.1.8 Id

You can use

```
id = #main // as a shortcut to Dom.select_id("main")
// or equivalently
id = #{"main"} // you can of course put an arbitrary expression
               // (of type string) inside the curly braces
```

This syntax can also be used in xhtml attributes values:

```
html = <div id=#main>some text</>
```

which is really a way of saying

```
html = <div id="main">some text</>
```

except that the syntax makes it clear what the string will be used for since the definition and usage of an id share the same syntax.

10.1.9 Actions

Opa features list of actions, which is a small dsl to transform the dom conveniently.
Note the syntax introduced below is really a structure, it does not execute anything. `Dom.transform` must be applied to a list of actions for the actions to be performed (so that you can build them of the server).

An action lists is just a list of actions, inside square braces and separated by commas (just like a list, except that it contains actions and not expressions). An action consist in a selector, a verb and an expression. The selector can be one of:

```
.some_static_class_name,
.{some_dynamic_class_name},
#some_static_id,
#{some_dynamic_id}
```

followed optionally by:

```
-> css // to select the style property
-> some_property // to select any static property (like style, value, etc.)
-> {e} // to select any dynamic property
```

The verb is either `<-` for setting the value, `+<-` for appending to the value or `-<-` for prepending to the value.

The expression is simply the value that will be set, appended to prepended to whatever is selected.
When the selector is not followed by `->`, the expression should be convertible to xhtml.
When the selector is followed by `-> value`, the expression should be convertible to string.
In all other cases, the expression must have type string.

Here is an instance of an action list, that replaces the content of the element pointed to by `#show_message` by a fragment of html.

```
do Dom.transform([#show_messages <- <>{failure}</>])
```

10.2 Client-server distribution

This section details the distribution between client and server.

10.2.1 Slicing

Opa is a language that can be executed both on the client and on the server, but at some point during the compilation process, it must be decided on which side does the code actually ends up, and where there are remote calls.

This is the job of the slicer.

The slicer can put each toplevel declaration (or component of a toplevel module) either on the server, or on the client, or on both sides. The slicer will not divide the code at a finer granularity.

The slicer can be told where a declaration should end up with the slicing annotations:

@server
: The declaration is on the server (but it does not mean that it will not be visible by the client)

@client
: The declaration is on the client (but it does not mean that it will not be visible by the server)

@both
: The declaration is on both sides. Because a declaration can do arbitrary side effects, there are two possible meanings:

  - Either the side effect is executed on both sides
  - Or the side effect is executed once (on the server) and the resulting value is shared between the two sides.

    By default, the slicer duplicate some side effects (printing for instance) and avoids to duplicate allocation of mutable structures. For instance:

    ```
    do println("Hello")
    ```

    will print "Hello" at toplevel on both sides. On the other hand

    ```
    s = Session.make(...)
    ```

    will create one unique session shared between the client and the server.

@both_implem
: This directive behaves the same way as `@both`, except that it explicitely forces the slicer to duplicate the declaration on both sides:

```
@both_implem s = Session.make(...)
```

This will create a session at the startup of the server and a session in each client.

Slicing annotations are not mandatory. When they are left out, the slicer decides where to place declarations: on both sides whenever it is possible, or on the only possible side when it has to.

When a slicing annotation is put on a (toplevel) module, it becomes the default slicing annotation for its components (and can be overriden by annotating the component with another annotation).

Now since everything cannot be executed on both sides, there are additional rules.
Primitives that are defined on one side can only be placed on this side.
When a primitive is server only, not only it is placed on the server, but it is implicitely tagged as server private, with the consequences explained below.

Whenever a declaration is tagged as server private, it cannot be called by the client (it is a slicing error), and any declaration using the current declaration becomes server private itself.
Since the tag server private propagate, there is a directive to stop the propagation: it essentially says that a declaration is now visible by the client (possibly after some authentication mechanism, checking the input, or simply because you have a server only primitive that does not really need to be private to begin with).
Note that a declaration that is server private can not be called by the client but can nevertheless call the client.

The relevant directives are:

@publish
Stops the propagation of the server private tags. Note that the declaration annotated as `@publish` are **not** the only entry point of the server:

```
@server f(x) = x
```

`f` can be called from the client.

@server_private
Initially, the only declarations that are tagged server private are the one containing server only primitives. This directive allows to tag some more declarations.

Finally, sometimes you will want to have a different behaviour on the server and on the client. This can be done, fairly simply, with `@sliced_expr`:

```
side = @sliced_expr({server="server"; client="client"})
do println(side)
```

This will print "server" on the server and "client" on the client.

@sliced_expr
This is simply a static switch between client and server. It can appear at any place in an expression.

Warning
The dependencies of the code are not analysed. As a consequence, trying to call the client from the server at the toplevel will slice correctly but will generate a runtime error (because there is no client yet, the server has not even been started yet).

10.2.2 Serialization

Any native opa value can be serialized: integers, floats, strings, records and functions.

Naturally, integers, floats, strings and records are copied when they are send to the other side. Since these structure are not mutable, this duplication is not observable.

Function serialization can be done in two ways:

- either the side receiving the serialized function builds a function that will make the remote call when applied
- or the side receiving the serialized function actually already has this function in its code and can call the local function instead of the remote function

The only remaining types are external types.
External types are not really serialized unless explicit serialization/deserialization functions are defined (with @serializer).
The default serialization generates an identifier and sends this identifier instead. When the side that generated the identifier unserializes it, it puts back the original structure in its place.
The only thing that is not possible in this design is to manipulate an external type from a side where it was not created. As a consequence, if an external type can be manipulated by primitives from both sides, then explicit serialization and deserialization functions **must** be given.

10.3 Including external files in an Opa server

The syntax defines two directives for including the contents of one file or the resources of one file:

- @static_content("foo.png") is replaced by a function that returns the content of compile-time foo.png ;
- @static_resource("foo.png") is replaced by a resource foo.png — with the appropriate last modification time, mime type, etc

Both directives support an additional argument for pre-processing the contents of the file before returning it.

Both directives have a counterpart that, instead of processing and returning one file, process a directory and return it as a stringmap:

- @static_content_directory("foo/") is replaced by a stringmap from file name to functions that maps key to the equivalent of @static_content(key). Of course, this stringmap is evaluated only once;
- @static_resource_directory is replaced by a stringmap from file name to functions that maps key to the equivalent of @static_resource(k
  Here, too, the stringmap is evaluated only once.

Again, these directives support an additional argument for pre-processing the contents of the file before returning it.

10.3.1 Examples

The two typical scenarios are embedding one resource:

```
handler = parser
  | "/favicon.ico"  -> @static_resource("img/favicon.ico")
```

and embedding many resources:

```
resources = @static_resource_directory("resources")
urls      = parser "/" -> start
          | resource={Server.resource_map(resources)} -> resource

server = Server.simple(urls)
```

For more details on parsers, see the related section For more details on Server.resource_map, see the library documentation.

Note
Relative paths are understood as starting from the project root.

10.3.2 See also

Resources embedded with these directives support runtime modification for debugging purposes. For more details, see the related section.

10.3.3 Runtime behavior

Release mode

Now, chances are that we want to secure these resources, e.g. to ensure that nobody will replace the nice MLstate logo with a not-quite-as-nice competitor logo. For this purpose, it is sufficient to compile your packages in release mode. A resource embedded by a package compiled in release mode is locked safely and can neither be dumped nor reloaded into the application by using --debug-* . Performance notes

All these directives are fast. Typically, @static_resource or @static_resource_directory will take a few milliseconds at start-up to determine whether they are executed in debug or non-debug mode, and there is no runtime performance loss in non-debug mode. When building resources, prefer these directives to @static_content or @static_content_directory are generally faster, as the final result will be a tad faster with @static_resource_*.

These directives interact nicely with zero-hit cache, provided that developers introduce resources in the zero-hit cache as follows:

```
resources = @static_resource_directory("resources")
urls      = parser "/" -> start
          | resource={Server.permanent_resource_map(resources)} -> resource

server = Server.simple(urls)
```

Of course, as usual with the zero-hit cache, you'll have to make sure that you are using URIs. For this purpose, as usual, you should take advantage of Resource.get_uri_of_permanent .

10.4 Index of keywords and directives

- @static_content
- @static_content_directory
- @static_resource
- @static_resource_directory

Chapter 11

The database

One of the unique characteristics of Opa is that the language contains a database. Database operations are fully integrated in the language, which increases code reusability, simplifies code coverage tests, and permits both automated optimizations and automated safety, security and sanity checks.

Whenever you launch an Opa, you do not need to worry about launching or connecting to a database, as Opa embeds its own database. If you launch several servers using the Opa launch scripts, Opa handles setup automatically and the servers transparently connect to each other to share their database.

In this chapter, we recapitulate the constructions of Opa dedicated to storage of data and manipulation of stored data.

11.1 General use

11.1.1 Overview

The core notion of the database is that of a *path*. A database *path* describes a position in the database and it is through a path that you can read, write, remove data, or access its history.

Declare a path with holding *one* value of a given type:

```
db /path/to/define : type
```

Note that the path is compulsory. Opa uses it at launch-time to verify that the database stored to the disk is compatible with the application. If the database schema has changed, Opa will offer the choice between proceeding to an automated database migration or leaving the application.

Read from said path:

```
x = /path/to/data
```

Modify the contents of said path:

```
/path/to/data <- x
```

Note that the database is structured. Therefore, you may define the following paths:

```
db /db/a: int
db /db/b: string
```

and then you access path `/db` with type `{ a: int; b: string }`.

Conversely, you may define

```
db /db2 : { a: int; b: string } / { c: float }
```

and then access paths `/db/a`, `/db/b` or `/db/c` directly.

Writing forces the state of the db. For instance, the following operation is valid:

```
/db2/c <- 3.14
```

If `db2` held values for `/db2/a` and `/db2/b`, both values are overwritten.

11.1.2 Default values

Each path has a *default value*. Whenever you attempt to read a value that does not exist, the default value is returned:

```
/db2 <- { a = 3; b = "four" }
x = /db2/c //x holds 0.0
```

While Opa can generally define a default value automatically, it is often a good idea to define an application-specific default value:

```
db /path/to/something = value
```

When you do not define a default value, Opa uses the following strategy:

- the default value for an integer is `0`;
- the default value for a floating-point number is `0.0`;
- the default value for a string is `""`;
- the default value for a record is the record of default values;
- the default value for a sum type is the case which looks most like an empty case (e.g. `{none}` for `option`, `{nil}` for list, etc.)

11.1.3 Native maps

Opa offers special treatment for values of type `stringmap` or `intmap`. If you define

```
/db3/mymap: intmap({ a: int; b: string })
```

a shortcut (and optimized) notation is available:

```
x = /db3/mymap[12]   //Whichever value is associated to 12 in the map, or a default value
y = /db3/mymap[12]/b //Field b of said value
```

11.1.4 Manipulating paths

Most operations deal with data, with the syntaxes presented above. Other operations deal with meta-data, that is the path itself, and require a slightly different syntax:

```
Db.modification_time(!/path/to/data)
```

A path written with notation `!/path/to/data` is called a *native paths*. Native paths represent a snapshot the state of the path. Should you write a new value at this path at a later stage, this will not affect the data or meta-data of these paths.

```
path = !/path/to/data          //Take a snapshot of the path
/path/to/data <- some_new_data //Modify data
Db.modification_time(path)     //The old modification time
```

Should you need to make reference to the *latest version* of data and metadata, you will need to use a *reference path*, as obtained with the following notation:

```
ref_path = @/path/to/data
```

Note that the write operation actually takes such a path, and the two following operations are actually equivalent:

```
/path/to/data  <- value
@/path/to/data <- value
```

Once we have defined a reference path, we may write

```
ref_path <- value
```

11.1.5 Manipulating databases

An Opa application can manipulate several databases, which may have distinct engines, or may be stored to distinct networks, for instance. This is a powerful tool for fine-tuning application performance or for ensuring that critical data is backed up.

For this purpose, you should give a name to the database and specify a default engine (at the time of this writing, either `local` or `remote`), as follows:

```
database @local("./where/to/store/")
database my_db = @remote(localhost:4849)
```

With this definition, any path starting with `/my_db` will be stored to database `my_db`. The default database is always named `database`.

Note that the choice of engine can be superseded by command-line options.

The full syntax is the following:

```
dbdef ::= database [id:] <spec>
spec ::= @meta | @local[("path")] | @remote[...]
```

11.1.6 Transactions

Whenever several users or several treatments need to access the database simultaneously, consistency needs to be enforced. For this purpose, Opa offers a mechanism of *transactions*. It's also more efficient, if you do some database operations in a row, to encapsulate them explicitely in a transaction.

In their simplest and most common form, transactions take the form of function `Db.transaction`:

```
atomic() =
(
  //...any operations
)

result = Db.transaction(atomic)
match result with
  | {none} -> //a conflict or another database error prevented the commit
  |~{some} -> //success, [some] contains the result of [atomic()]
```

It is possible to get much finer control over what is done with a transaction ; unlike most common database engines, Opa doesn't force a transaction to be run in one block: it can be suspended and continued later, without blocking the database in any way.

```
tr = Transaction.new()

do tr.in(atomic)

// some other treatments

match tr.commit() with
  | {success} -> // ...
  | {failure} -> // ...
```

Here, only the `atomic` function is run within the transaction, the other treatments at the top level will be done normally. This means that, until the transaction `tr` is committed, its results aren't visible to the outside. Moreover, operations executed in `tr` won't see the changes done outside, which ensures that it proceeds in a consistent database state. There is no limit to the number of `tr.in` you can do in the same transaction.

The problem with this approach is that the operations done on both sides could conflict, and `tr` could stop being valid because of changes written to the database in the meantime. This is why `commit` can return a `failure`, which can be used to either try again or inform the user of the error.

The continuable transactions are quite useful in a web application context: they can be used to write operations done by a user synchronously, then only commit when he chooses to validate. You can get back data from a running transaction with `tr.try`, by providing an error-case handler:

```
tr = Transaction.new()

some_operations() = some(/* ... */)
error_case() = none

r = tr.try(some_operations, error_case)
```

If you are using multiple databases, the commit of a transaction is guaranteed to be consistent on all the ones that were accessed in its course (if the commit fails on a single database, no database will be modified). However, when using `Transaction.new()`, a low-level transaction is only started on each database as needed: if you want to make sure your transaction is started at the same point on different databases, use `Transaction.new_on([database1,database2])` instead.

Chapter 12

Running Executables

12.1 The compiler : opa

This chapter describes the *opa* compiler, which compiles Opa source files to package object files and links these object files to produce standalone executables.

12.1.1 Level and Modes of utilisation

The compiler offers 2 levels of utilisation, a high-level one, and a low-level one, and 2 low-level modes of execution, the compilation and the linking.

12.1.1.1 Standard utilisation

The high-level utilisation of *opa* is an *autobuild* mode. It offers an automatic way for building medium and large applications composed of several packages. It simplifies the set up of a build system. This is the default behavior of *opa*, able to build an application as a standalone executable, from all its source files, recompiling only what is needed, without needing e.g. a Makefile. During the same call to opa, the compiler will process to the compilation of all needed packages, and perform the final linking at the end.

Same example:

```
user@computer:~/$ ls
foo.opa

user@computer:~/$ opa foo.opa

user@computer:~/$ ls
foo.exe  _build  foo.opa  foo.opx
```

12.1.1.2 Other modes

The low-level utilisation of *opa* is comparable with standard compilers like *gcc* or *java*. It takes one of more opa files from the same package, compile them into compiled packages, or link previous compiled packages and opa files, depending on the activated mode.

12.1.1.3 Compilation

This mode corresponds to the compilation of the files composing a package into intermediate object files, ready to be linked. Note than compiled packages can be shared between linked applications. For example, the distribution of Opa contains the compiled packages of the stdlib (cf opx files), and these opx files are linked with all Opa applications.

12.1.1.4 Linking

The linking is the final step in the build process. This produces an standalone executable from given previously compiled *opx* files.

About linking
It is possible to give some opa file for the linking. In this case, they are considered to be all regrouped in the same package. This can be used for a quick prototyping, and small applications, but this practice is discouraged for medium and big applications. Use rather autobuild mode (default) or link only packages previously compiled (*opx*)

12.1.1.5 Example

Example:

```
user@computer:~/$ ls
foo.opa

user@computer:~/$ cat foo.opa
package foo
do print("this is foo.opa\n")

user@computer:~/$ opa -c foo.opa

user@computer:~/$ ls
foo.opa  foo.opx

user@computer:~/$ opa -l foo.opx

user@computer:~/$ ls
a.exe  _build  foo.opa  foo.opx

user@computer:~/$ ./a.exe
this is foo.opa
```

12.1.2 Arguments, input

The compiler recognizes a certain number of file extension in its command line. For more details about the file extension, see the section [Filename extensions]

extension	action
*.conf	read a conf file for package organisation
*.cmx, *.cmxa	add ocaml native libraries for linking
*.opa	compile and/or link opa files
*.opack	read option and arguments from an external file
*.opp	link with an Opa plugin
*.opx	link with a previous compiled package

12.1.3 Output of the compiler

The compiler produces a number of files and/or executable, depending on its mode of utilisation, and the options activated.

output	mode or option	description
api	--api	used by *opadoc*
executable	linking or autobuild	standalone executables

output	mode or option	description
odep	--odep	dependency graphs of the compiled application
opx	compilation or autobuild	compiled packages

12.1.4 Options

This list contains only the offical options supported in the distributions of Opa. For more details about your specific opa version, use *opa --help*

12.1.4.1 Levels and modes of utilisation

option	argument	description
<default>	none	High-level compilation mode, compiles everything and build a standalone executable
-c	none	Low-level compilation mode, compiles the current package
-l	none	Low-level linking mode, link the given packages and opa files
--autocompile	none	High-level mode, compile every package, but do not link at end, for building e.g. shared opa librairies

12.1.5 Warnings

The warnings of the compiler are organized in hierarchical classes composing a warning tree. Each warning class is a node of this tree. Each class can contain sub-classes (children), and can be switched into :

- ignored
- warning
- error

It is possible to change the default properties of a warning class using the following options:

option	description
--no-warn	ignored (not printed)
--warn, --no-warn-error	printed as a warning, the compilation continues
--warn-error	treated as an error, the compilation stops

For accessing to the list of all warnings, use *--warn-help*.

The name of the classes are implied by their place in the warning tree, this is the path of nodes leading to the class, lowercased, and separated by dots. Example, the class *coding* corresponds to rules about coding style, and contains a subclass named *deprecated* used for pointing out the use of deprecated constructions. So, the name of this subclass for the command line is *coding.deprecated* :

```
user@computer:~/$ opa --warn-error coding.deprecated foo.opa
```

A property set to a class is applied to all children of the class. For example, *--warn-error coding* implies *--warn-error coding.deprecated*

About the root warning class
All warning classes have a common ancestor named *root*, this is the root of the tree. You can use e.g. *--warn-error root* for enabling all warnings as errors. Although this is the root of the tree, the name *root* does not appear in each warning class name (this is the only exception)

Example:

```
user@computer:~/$ ls
foo.opa

user@computer:~/$ cat foo.opa
f(x) = 0

user@computer:~/$ opa foo.opa
Warning unused
File "foo.opa", line 1, characters 2-3, (1:2-1:3 | 2-3)
  Unused variable x.

user@computer:~/$ opa --no-warn unused foo.opa

user@computer:~/$ opa --warn-error unused foo.opa
Warning unused
File "foo.opa", line 1, characters 2-3, (1:2-1:3 | 2-3)
  Unused variable x.

Error
Fatal warning: 'unused'
```

12.2 opadoc

This executable is a documentation generator for Opa projects. It reads special comments from Opa files documenting functions and type declarations, and produces various formats of output (html, man pages, etc.)

12.2.1 Arguments, input

The documentation is generated from

- *opa* files documented with special comments using the *opadoc* syntax
- *api* files generated by *opa --api*, containing inferred type informations

The tool parses all the given files, computes an association between types and comments, then generates a full documention.

Argument should be :

- opa files directly ;
- directories containing opa files

Caution
api files are not given in the command line of *opadoc*, but found automatically near *opa* files or in the given directories

12.2.2 Output

Formats of output can be specified. Currently, only an *html* mode is implemented. Contributions to support more formats are welcome.

At the end of the file analysis and association (*opa* + *api*) it is possible to store the result of this association, in an *apix* file

12.2.2.1 html

A directory is generated containing several html files, with a list of types, values, packages, etc.

12.2.3 Options

option	argument	description
--output-dir, -o	<dir>	Specify an Output directory where to put the generated documentation. The default value is *doc*
--private	none	Export private and abstract types/values. This option can be used during e.g. the co-developpement of a package, internally. Note that the documentation of the stdlib in the distribution of Opa is not compiled with this option.

12.2.4 Example

```
user@computer:~/$ ls
bar  foo.opa

user@computer:~/$ ls bar
bar.opa

user@computer:~/$ opa --api bar/bar.opa foo.opa

user@computer:~/$ ls
bar  foo.api  foo.opa  foo.exe

user@computer:~/$ ls bar
bar.api  bar.opa

user@computer:~/$ opadoc foo.opa bar -o doc

user@computer:~/$ open doc/index.html
```

12.3 opa-plugin-builder

Coming soon…

12.4 opa-plugin-browser

Coming soon…

12.5 Opa applications, at run-time

About this section

This section details the use of applications built with Opa, including:

- command-line arguments;
- logging of events;
- network administration;
- environment variables.

12.5.1 Accessing privileged system resources

When developing your application, it is perfectly acceptable (and even recommended) to test it on user-allowed ports, as the default port 8080. However, when your application is ready and you want to deploy it and show it to the world, you will probably need to allow it to use port 80, as well as some other privileged ports, depending on your application.

There are basically two ways to do that:

- Run your application with the root account. This will work, as with any other user, and Opa will not attempt to drop privileges. Although we did our best to make Opa as secure as possible, and you certainly did the same for your application, it is a bit uncomfortable to run a full application with administrative rights. Consequently, we do not advise this solution.
- Run your application in user-land, and handle privileged actions with specific tools. This is much safer and often more flexible. There are at least two very different ways to do that:
  - use authbind to allow your application to access directly a privileged port;
  - or put a priviledged dispatcher in front of your application (e.g. HAProxy or Nginx).

12.5.2 Debugging resources

Opa applicaions support the following command-line options, which can be used to make generated files editable at runtime:

- --debug-editable-js makes the compiled JS editable at runtime;
- --debug-editable-css makes the compiled CSS editable at runtime;
- --debug-editable-file f makes embedded file f editable at runtime;
- --debug-editable-all makes everything editable at runtime;
- --debug-list-resources lists the resources that can be made editable.

Each of these options creates a directory *'opa-debug'*, which contains all the editable files. If a file (other than JS) is already present, it is reused instead of the embedded file. Otherwise, the file is created with the contents embedded in the executable. Now, if the file is modified, it is automatically reloaded (without having to relaunch the server) and is immediately visible on the client.

By the way, if a debug file is removed during the execution of the server, this file is automatically recreated, without having to relaunch the server. We also log any change to the application logs.

Of course, the file is never saved back into the executable. You'll need recompilation for this kind of thing.

12.5.3 Index of command-line arguments

Appendix A

Filename extensions

This appendix lists the different filename extensions you can meet in an Opa project, with a short description of the nature of the corresponding file or directory.

This part describes for each kind of file in an Opa project :

- a description of the file, meaning what kind of file it is, and what it contains ;
- an output section, documenting what tool/application produces these files ;
- an input section, documenting what tool or what application actually reads these files

A.1 api

A.1.1 Description

api files are text files using a *json* syntax for encoding the types informations extracted from Opa files, associated to their positions in the source code.

A.1.2 Produced by

For each opa file, there can be one corresponding api file, generated by the opa compiler using the option *--api*

```
user@computer:~/$ ls
bar  foo.opa

user@computer:~/$ ls bar
bar.opa

user@computer:~/$ opa --api bar/bar.opa foo.opa

user@computer:~/$ ls
bar  foo.api  foo.opa  foo.exe

user@computer:~/$ ls bar
bar.api  bar.opa
```

A.1.3 Used by

These files are read by *opadoc* to build code documentation. They cannot be given in the command line of opadoc directly, but they are found automatically by association with any Opa file found directly or in one of the subdirectories of the commnand line :

```
user@computer:~/$ ls
bar  foo.api  foo.opa

user@computer:~/$ ls bar
bar.api  bar.opa

user@computer:~/$ opadoc foo.opa bar -o doc
```

A.2 api-txt

A.2.1 Description

These files are text files written using the Opa syntax, containing the inferred types of toplevel elements of a corresponding Opa file. They contain a subset of the information contained in a *api* file, but in an human readable syntax (opa rather than json).

A.2.2 Produced by

Same than *api* files, generated by *opa* with the option *--api*

A.2.3 Used by

Meant to be read by an human, e.g. for debugging.

A.3 conf

A.3.1 Description

Configuration of packages for building medium and large applications. These files are used for setting the package organisation of an application, without editing the source files. Their utilisation is optional. It offers a functionality equivalent with the keyword **import** and **package** of the Opa syntax.

A.3.2 Produced by

Hand written by the author of an opa application.

A.3.3 Input

opa

A.3.4 Syntax

Commented line starts with a sharp character #. Then it follows the entry point *packages* of this grammar :

```
:: packages <- package list

package <- $package_name ":" entry list

entry <- import / source

import <- "import" package_entry

source <- $filename

package_entry <- $package_name / $extension_package_name
```

About filename

- Relative path of filenames are given from the emplacement of the conf file itself
- It is possible to refer to *environment variables* in a conf file

---

About package_name
This follows the same convention than the keyword *import* in the opa syntax, but no space are allowed using the extension syntax, as in these authorized examples:

```
import stdlib.*
import toto.{foo,bar}
```

Example

Here is a small example:

```
# a first package, with 2 files
my_package:
  import toto.{foo,bar}
  relative/path/to/myfile.opa
  relative/path/to/myotherfile.opa

# a second package, importing the first one
my_otherpackage:
  import my_package
  path/to/some_file.opa
```

A.4 jsconf

A.4.1 Description

conf for bsl js files, Cf part about plugin for the syntax of these files

A.4.2 Produced by

Hand written by the author of a js plugin.

A.4.3 Used by

opa-plugin-builder

A.5 opa

A.5.1 Description

This is the extension of the Opa files. Most of the files in an Opa project are .opa files

A.5.2 Produced by

Hand written by the author of an Opa application.

A.5.3 Used by

opa, *opadoc*

A.6 opack

A.6.1 Description

opack files are used to regroup the command line options and arguments for invoking *opa* to build an Opa application.

Example:

```
user@computer:~/$ opa myproject.opack
```

is almost equivalent as

```
user@computer:~/$ cat myproject.opack | grep -v '#' | xargs opa
```

A.6.2 Produced by

Hand written by the author of an Opa application.

A.6.3 Used by

opa

A.6.4 Syntax

This is a file where lines correspond to argument or options of the opa compiler. Lines may be commented with #, and it is possible to refer to *environment variables* in a conf file

A.6.4.1 Example

```
# This is an example of opack file
myfile_1.opa
myfile_2.opa
--warn-error root
```

A.7 opp

A.7.1 Description

opp stands for OPa Plugin. An *opp* is a directory containing object files and compiled code, for building an Opa application using external primitives written directly in *Javascript* and/or in *Ocaml*.

A.7.2 Produced by

opa-plugin-builder

A.7.3 Used by

opa, *opa-plugin-browser*

A.8 opx

A.8.1 Description

An *opx* is a directory containing object files and compiled code from a Opa package. For each Opa package correspond one *opx* directory once compiled by *opa*. Some *opx* files are distributed with *opa*, these are the compiled packages of the standard library.

A.8.2 Produced by

opa

A.8.3 Used by

opa

Appendix B

Bindings with other languages

It is possible to bind Opa with other languages thanks to a **BSL**: Binding System Library. Currently, there is a native support to Javascript on the client side, and to OCaml on the server side.

This chapter shows an example of binding between Opa, as well as an example to see C primitives calls from Opa.

A complete example of binding with an external Javascript library is available in the chapter hello recaptcha.

In Opa, a function written in an other language is called a foreign function, a *bypass*, or an *external primitive*. All this designations mean that the implementation of the function is not written in Opa.

Part of the Opa standard library uses external primitives, such as low level arithmetic operations, bindings with the DOM API on the client side, etc.

Using external primitives in Opa applications has two main motivations :

- provide a new functionality not available in the standard library, and the implementation cannot easily be written in Opa (e.g. using low level system call). Although it does not often happen because of the completeness of the Opa standard library, it remains useful for specific applications.
- bind with existing code: you want to use an external library without reimplementing it in Opa (e.g. numerical computing library), or waiting for it to be rewritten in Opa (e.g. for a quick prototyping, hybrid applications)

Since *OCaml* supports bindings for many languages, it is possible, on the server side of an Opa applications, to use functions written in any language supporting an OCaml bindings (C, Java, Python, etc.)

Although we are working on a native support for binding Opa and *C* applications, this is not available in the distribution of Opa yet. Waiting for this support, it is nevertheless possible to bind Opa and *C* via *OCaml* (This chapter contains an example using a external primitive implemented in *C*, using the *OCaml/C* binding)

Client/Server distribution
When you define an external primitive, the function is available only on the side of the implementation. (Javascript on the client side, *OCaml*, *C* on the server side). If you do not provide both client and server sides of the implementation of the external primitive, it has an impact on the client/server distribution of the Opa code. You have to think about it before deciding to use an external binding. Writing a function in pure Opa gives you the guarantee that the function is available on both side.

B.1 Hello binding

This part shows a complete example, including compilation command line.

B.1.1 Look at the files

```
user@computer:~/$ ls
hello_binding.opa  plugin.js  plugin.ml  Makefile
```

Makefile

```
all : plugin opa

plugin:
        $(OPA_PLUGIN_BUILDER) -o plugin plugin.js plugin.ml

opa:
        $(OPA) plugin.opp hello_bindings.opa
        @echo "run: ./hello_bindings.exe"

clean:
        rm -rf hello_bindings.exe
        rm -rf _build _tracks
        rm -rf plugin.opp
```

B.1.2 External implementation

Implementations goes into files, and file names should be called the same way between server and client side implementation.

The implementation files are files written in the syntax of the bound language, adding some Opa directives for registering the primitives and the types exported for being used from Opa code.

In Javascript, we use a special syntax for introducing function names (*args*), used for simulating namespaces. Using the directive *args* following a *register* directive defining the function *stammer* in the file *plugin* will introduce the function named *plugin_stammer*. The use of *args* is optional, we can write instead:

```
(* This is an Ocaml file, containing opa preprocessing directives *)

##register stammer : string -> string
let stammer s = "server is stammering: " ^ s ^ s ^ s
```

```
// This is an Javascript file, containing opa preprocessing directives

##register stammer : string -> string
##args(s)
{
    // this is a standard javascript code,
    // we are there in the body of a function, with an argument named 's'
    return "client is stammering: " + s + s + s ;
}
```

```
##register stammer : string -> string
function plugin_stammer(s)
{
    return "client is stammering: " + s + s + s ;
}
```

but it is not advised to handle manually the namespaces. Using the directive *##args* implies a check of arity with the registered type.

B.1.3 Plugin Compilation

The plugin files are compiled using the tool opa-plugin-builder, distributed with the *Opa* compiler. You can refer to the documentation of the tool for more details, and the part about opp files. *opa-plugin-builder* produces an *opp* directory from plugin sources files. The name of the plugin should be specified with the option *-o* :

```
user@computer:~/$ opa-plugin-builder plugin.js plugin.ml -o plugin
user@computer:~/$ ls
hello_binding.opa  plugin.js  plugin.ml  plugin.opp
```

B.1.4 Plugin browsing

The plugin *plugin.opp* is ready to be linked with an Opa application. You can browse the plugin, using the tool opa-plugin-browser

```
user@computer:~/$ opa-plugin-browser plugin.opp
(you enter a tiny shell-like environment)
bslbrowser:/$ ls
  + module <plugin>
bslbrowser:/$ cd plugin
bslbrowser:plugin/$ ls
+ in module <plugin> \:
        stammer            \: string -> string {js, ml}
(try also: 'help')
```

When you are browsing a plugin, you can check that the exported functions are correctly stored in the *opp*, and get familiar with the information stored about the external primitives (try *ls -v*).

B.1.4.1 Key normalization

External Primitives are indexed using a key normalisation. The key is obtained from the path (file of declaration + module hierarchy), by concatenation of all path elements, using *underscore* between elements replacing dots by *underscores*, and finally converted to lowercase.

In this example, the key of the function *stammer* is *plugin_stammer*. This key is used for referring to this function from the Opa code.

For implementing primitives on both sides, it is mendatory for the server and the client primitives to have the same name. This implies to choose valid filenames for Ocaml modules. For files implementing primitives on the client only, it is possible to use dots and dash in filenames, their are transformed for computing the key : "-" is replaced by "_dash_" and "." is replaced by "_dot_" .

Example: "jquery-1.6.1.js" becomes : "jquery_dash_1_dot_6_dot_1.js"

B.1.5 Binding in Opa

The syntax for introducing external primitives in Opa is *%% key %%*, referring to the *key* of the external primitive.

```
plugin_stammer = %% plugin_stammer %%
```

Tip
The keys found in the syntax are also normalized by the compiler, which means that calls are case insensitive, and so the following lines are equivalent:

```
plugin_stammer = %%plugin.stammer%%

plugin_stammer = %% plugin.stammer %%

plugin_stammer = %% Plugin.stammer %%

plugin_stammer = %% pLuGin_staMmeR %%
```

You can find there a simple *Opa* application for calling either the client or the server side implementation of the function *stammer*, with an input for giving the argument.

```
plugin_stammer = %%plugin.stammer%%

@server stammer_server = plugin_stammer
@client stammer_client = plugin_stammer

stammer(a, side) =
  match side with
  | {client} -> stammer_client(a)
  | {server} -> stammer_server(a)

action(side) =
  a = Dom.get_value(#input_a)
  Dom.transform([ #result <- stammer(a, side) ])

button(id, message, side) =
  <a id={id:string}
     class="button"
     ref="#"
     onclick={_->action(side)}>{message:string}
  </a>

page() =
  <>
  <h1>Hello Bindings</h1>
  <h2>Argument</h2>
  <input id="input_a"/>
  <h2>Result</h2>
  <div id="result" />
  <br/>
  <h2>Calling functions (using argument input)</h2>
  {button("stammer_client", "Compute stammer on the client", {client})}<br/>
  {button("stammer_server", "Compute stammer on the server", {server})}<br/>
  </>

server = one_page_server("Hello Bindings", page)
```

Compiling the application requires an extra argument to be passed to the Opa compiler so that it finds the type of the external primitive, and find the implementation to link with the server. Note that parameters are not ordered.

```
user@computer:~/$ opa plugin.opp hell_bindings.opa

user@computer:~/$ ./hell_bindings.exe
```

B.2 Hello binding C

This part shows an integration of C code in an Opa application, via an *OCaml* binding. A native support is currently in integration, available in a future version of Opa.

B.2.1 Look at the files

```
user@computer:~/$ ls
freq.opa  freq.c  c_binding.ml  Makefile
```

Makefile

```
DIR:=$(shell pwd)

all: freq.o plugin opa

freq.o: freq.c
        $(OCAMLOPT) freq.c

plugin:
        $(OPA_PLUGIN_BUILDER) -o c_binding c_binding.ml

opa:
        $(OPA) --mllopt $(DIR)/freq.o c_binding.opp freq.opa
        @echo "run: ./freq.exe"

clean:
        rm -rf c_binding.opp
        rm -rf freq.exe freq.o
        rm -rf _build _tracks
```

```
#include <stdio.h>
#include <sys/time.h>
#include <string.h>
#include <unistd.h>

unsigned long long int rdtsc(void)
{
    unsigned long long int x;
    __asm__ volatile (".byte 0x0f, 0x31" : "=A" (x));
    return x;
}

int freq()
{
    struct timezone tz;
    struct timeval tvstart, tvstop;
    unsigned long long int cycles[2];
    unsigned long microseconds;
    int mhz;

    memset(&tz, 0, sizeof(tz));

    gettimeofday(&tvstart, &tz);
    cycles[0] = rdtsc();
    gettimeofday(&tvstart, &tz);

    usleep(250000);

    gettimeofday(&tvstop, &tz);
    cycles[1] = rdtsc();
    gettimeofday(&tvstop, &tz);

    microseconds = ((tvstop.tv_sec-tvstart.tv_sec)*1000000) + (tvstop.tv_usec-tvstart. ↩
        tv_usec);
```

```
    mhz = (int) (cycles[1]-cycles[0]) / microseconds;

    //        printf("%i MHz\n",mhz);

    return mhz;
}

// OCAML binding
#include <errno.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/alloc.h>
#include <caml/compatibility.h>
#include <caml/fail.h>

value freq_get()
{
    CAMLparam0();
    CAMLreturn(Val_int(freq()));
}
```

```
##register freq_get : -> int
external freq_get : unit -> int = "freq_get"
```

```
freq_get = %%c_binding.freq_get%%
do print("freq : {freq_get()} Mhz\n")
```

B.2.2 Compilation

```
user@computer:~/$ ocamlopt.opt freq.c
user@computer:~/$ opa-plugin-builder c_binding.ml -o c_binding
user@computer:~/$ opa --mllopt $PWD/freq.o c_binding.opp freq.opa
user@computer:~/$ ./freq.exe
```

For more details about integration between C/OCaml, you can refer to the manual of Objective Caml.

B.3 Documentation

This paragraph describes the exhaustive list of directives handled by *opa-plubin-builder* in OCaml and Javascript files. The conventions adopted to describe the syntax of theses directives are the same as the one used for describing the core language syntax.

The directives are used to declare the types and values to export in Opa. Opa is type-checked at compile time, hence it is mandatory to declare the types of the external primitives.

Caution
There is a restriction about types available in the interface between Opa and foreign implementations. You should keep in mind that there are differences of representation of values between languages, a fortiori between Opa, OCaml, and Javascript, and a value passed through the register interface should most of the time be translated. For the most common types, an automatic projection is done by Opa (constants, bool, option). For more complex values, you will need to handle the projection manually, via the ServerLib on the server side, and the ClientLib on the server side. When you are designing a new plugin, you should keep in mind that value projections may be inefficient, and it often is more convenient to use primitive types only.

B.3.1 General syntax

B.3.1.1 Syntax of types

```
type ::=
  | int
  | string
  | float
  | void
  | bool
  | option(<type>)
  | <type>* sep , -> <type>
  | <type-var>
  | <type-ident> ( <type>* sep , )?
  | opa[<type>]
  | ( <type> )
```

B.3.1.2 Syntax of directives

```
directive ::= ## <properties>? <directive_name> <arguments>

properties ::= [ <property>+ sep , ]
property ::=
  | <tag>
  | <tag> : <value>

tag ::= <see tags documentation, depends of the directive_name>

directive_name ::=
  | extern-type
  | opa-type
  | module
  | endmodule
  | register
  | args
  | property

arguments ::= <see arguments documentation, depends of the directive_name>
```

The properties are used for setting options to directives, for modifying their behavior, or for attaching extra information to them. The directive *##property* is used for changing the behavior of the other directives until the end of the current file. The supported tags are specific to the directive. In standard cases, you should not need tags and properties (advanced use only)

```
verbatim-ocaml ::= some valid OCaml code

type-ident ::= same as Opa type ident

parameters ::=
  | ( <type-var>+ sep , )
```

Syntax of arguments for ##extern-type

```
arguments(##extern-type) ::=
  | <type-ident> <parameters>? ( = <verbatim-ocaml> )?
```

Examples:

```
##extern-type hashset('a) = ('a, unit) Hashtbl.t
##extern-type black
```

```
arguments(##opa-type) ::=
  | <type-ident> <parameters>?
```

Examples:

```
##opa-type option('a)
```

```
arguments(##register) ::=
  | <name> : <type>
  | <name> \ <name> : <type>
  | <name> \ `<verbatim-ocaml>` : <type>
```

Examples:

```
##register int_of_string : string -> option(int)
##register int_add \ `Pervasives.( + )` : int, int -> int
```

Tip
Backslash \ can also be used to dispatch a directive on several lines if it does not fit on one line of source, e.g. a function using big types.

Example:

```
##register my_function :          \
  (int, int, int, int -> int),   \
  int ->                         \
  int
```

B.3.1.3 Tags

Tags are needed for advanced uses only. The casual user can simply ignore them.

directive name	supported tags	arguments
extern-type	opacapi	type definition
opa-type	opacapi	type definition
module	-	module name
endmodule	-	-
register	backend, opacapi, restricted, no-projection, cps-bypass	implementation name, and type of the value
args	-	formal parameters
const	-	-
property	mli, endmli	-

B.3.2 About Supported Types

Tip
Compiling OCaml plugins, *opa-plugin-builder* will perform a type check to ensure that the functions defined in plugins have consistent types with respect to their registered definition. We are also working on a step of Javascript validation to ensure some consistent properties (functions arities, etc.)

Example: the following definitions will generate a static error :

```
##register foo : int, int -> int
let foo a b = a +. b
```

```
##register foo : int, int -> int
##args(x)
{
  return x+x;
}
```

The supported types in the interface are :

- void and constants;
- bool;
- option;
- external types;
- opa types;
- polymorphic types.

Before writing a new plugin, you should think about the kind of types you will need to manipulate, and the side where the values of theses types will be processed: server only, client only, or both. The latter case requires types to be serializable.

Caution
You should also try not to use functional arguments. The Opa code is rewritten in CPS, and the external function are generally not in CPS. Opa proceed to a projection of a CPS function into a non CPS function for passing it to the external functional primitive, using an hint about the time needed for this function to compute. In general it is not possible to know statically a bound of the time needed for a functional argument to compute : typically, in case of an asynchronous function (e.g. with client/server I/O) you may get a runtime error when the external functional primitive will be executed, meaning that the response of the function is not available at the end of the timeout used in the projection. Nevertheless, if you really need to have asynchronous functional arguments, you can contact MLstate for a support, or browse the source code of the standard library (looking for the tag *cps-bypass*)

B.3.2.1 Automatic projections

As often when you write a binding between several languages, you have to deal with the fact that values can have different runtime representations in each language. The approach used e.g. in the OCaml/C binding, is to manipulated OCaml values using a special set of macros and function (*CAMLparam*, *CAMLreturn*, etc.), and to use a manual projection of values between the two languages, e.g. using *Val_int* for building an OCaml int from a C int, etc.

```
value add_ocaml_int(value a, value b)
{
  int ia, ib;
  CAMLparam2 (a, b);
  ia = Int_val(a);
  ib = Int_val(b);
  CAMLreturn (Val_int(ia + iab));
}
```

The code resulting of this manual projection is quite verbose, and the experience has shown that it is easy to write incorrect code, e.g. forgetting part of projections, especially if the code is not typed.

From this constatation, we decided in Opa to handle automatically the projections of values for the most common types allowed in the interface between Opa and foreign languages.

Let's see an example:

```
(*
  In the ServerLib, we have the following manipulation of int values :

  type ty_int

  val wrap_int : int -> ty_int
  val unwrap_int : ty_int -> int
*)
let add a b =
  let ia = ServerLib.unwrap_int a in
  let ib = ServerLib.unwrap_int b in
  let ic = ia + ib in
  ServerLib.wrap_int ic
```

This example implements a function adding two opa ints using the OCaml + primitive combined with projection between OCaml ints and Opa ints.

Now, we will save the code into the file *int.ml*, try to register this function and building an Opa plugin.

```
##register add : int, int -> int
let add a b =
  let ia = ServerLib.unwrap_int a in
  let ib = ServerLib.unwrap_int b in
  let ic = ia + ib in
  ServerLib.wrap_int ic
```

Let's try to compile it:

```
user@computer:~/$ opa-plugin-builder int.ml -o int
File "intMLRuntime.ml", line 1, characters 0-1:
Error: The implementation intMLRuntime.ml
       does not match the interface intMLRuntime.cmi:
       Modules do not match:
         sig
           val add : ServerLib.ty_int -> ServerLib.ty_int -> ServerLib.ty_int
         end
       is not included in
         sig val add : int -> int -> int end
       Values do not match:
         val add : ServerLib.ty_int -> ServerLib.ty_int -> ServerLib.ty_int
       is not included in
         val add : int -> int -> int
```

We get there an error, the code does not type. Why ?

The function manipulates int, and this is pretty clear that if you want to register a function working on int, you want to use it in Opa, so you will need to projects values. To avoid this, *opa_plugin_builder* transforms the code and inserts the projections, unless you specify in the interface that you are explictly manipulating opa values, and want to handle the projection yourself :

```
##register add : opa[int], opa[int] -> opa[int]
let add a b =
  let ia = ServerLib.unwrap_int a in
  let ib = ServerLib.unwrap_int b in
  let ic = ia + ib in
  ServerLib.wrap_int ic
```

```
user@computer:~/$ opa-plugin-builder int.ml -o int
# compilation is successfull
```

```
##register add : int, int -> int
let add a b = a + b
```

```
user@computer:~/$ opa-plugin-builder int.ml -o int
# compilation is successfull
```

Tip
If the version of the compiler knows that some types have the same representation in Opa or in Ocaml, projections are not be generated.

B.3.2.2 Representation of values

Table B.1: OCaml

type	interface syntax	ocaml representation	opa representation
int	int VS opa[int]	int	ServerLib.ty_int
float	float VS opa[float]	float	ServerLib.ty_float
string	string VS opa[string]	string	ServerLib.ty_string
void	void VS opa[void]	unit	ServerLib.ty_void
bool	bool VS opa[bool]	bool	ServerLib.ty_bool
option	option('a) VS opa[option('a)]	'a option	'a ServerLib.ty_option

Table B.2: Javascript

type	interface syntax	ocaml representation	opa representation
int	int VS opa[int]	int	ClientLib_ty_int
float	float VS opa[float]	float	ClientLib_ty_float
string	string VS opa[string]	string	ClientLib_ty_string
void	void VS opa[void]	undefined or null	ClientLib_ty_void
bool	bool VS opa[bool]	bool	ClientLib_ty_bool
option	option('a) VS opa[option('a)]	None if null or Some 'a otherwise	'a ClientLib_ty_option

An API is available for manipulating opa values, on the server side as well as on the client side.

B.3.2.3 External type VS Opa type

If you want to use an external library providing an API manipulating types such than you do not need to see in Opa the implementation of these types, you can refer to the documentation of external types.

If you want to be able to manipulate Opa values in the external implementation, you can refer to the documentation of Opa types.

B.3.3 External types

External types are used for manipulating in Opa foreign values without manipulating the raw implementation of their types, using a foreign interface. Once in Opa, each external type is unifiable only with itself. The implementation is hidden in Opa.

An extern type is generally available only in one side : the side of its implementation. Nevertheless, it is possible to define a common API on the client and the server side, and providing explicit (de)serialization functions to make types available on both side, and able to travel from one side to the other.

Tip
In the standard library, low level arrays are implemented using extern-type definitions.

B.3.3.1 Complete example using external types

This example shows how to use the module *Big_int* of the *OCaml* standard library to add 2 bigints given by a user via a simple web interface.

```
user@computer:~/$ ls
bigint.ml  bigint.opa  Makefile
```

Makefile

```
all : plugin opa

plugin:
        $(OPA_PLUGIN_BUILDER) -o bigint bigint.ml

opa:
        $(OPA) bigint.opp bigint.opa

clean:
        rm -rf _build _tracks bigint.opp bigint.exe *.log
```

bigint.ml

```
(**
   Minimal Opa Binding for the Big_int module (Opa Manual)
*)

(**
   External type definition.
   The left part is the name of the type as it will be available in Opa.
   The right part is the name of the type as it is defined by the Ocaml library
*)
##extern-type BigInt.t = Big_int.big_int

(**
   For this example, we will export a reduced API, just to add 2 big int
   given by the user of a very simple web page.
*)

(**
   This function exists already 'as is' in the Ocaml library,
   we can use the \ ` ` syntax : the function will be inlined in
   the generated code.
*)
##register add \ `Big_int.add_big_int` : BigInt.t, BigInt.t -> BigInt.t

(**
   We prefer to deal with option for handling error cases in Opa.
*)
##register of_string : string -> option(BigInt.t)
let of_string string =
  try
    Some (Big_int.big_int_of_string string)
  with
  | _ -> None

(**
```

```
   And we want to see the result
*)
##register to_string \ `Big_int.string_of_big_int` : BigInt.t -> string
```

bigint.opa

```
/**
 * Simple server using big int
**/

/**
 * {1 Binding}
**/

/**
 * The type definition in opa.
 * The name should be the same as the one used in the plugin
 * The keywork 'external' means that the implementation is not
 * available in Opa, and values of this type can only be manipulated
 * via the API of the plugin.
**/
type BigInt.t = external

BigInt = {{

  /**
   * The syntax for binding foreign functions is %% key %%
  **/
  add = %% bigint.add %%
  to_string = %% bigint.to_string %%

  /**
   * For lisibility of the code, we can add types coercion
  **/
  of_string = %% bigint.of_string %% : string -> option(BigInt.t)

}}

/**
 * {1 Web page}
**/

/**
 * Compute the addition, or return a hint message in case of error
**/
@publish compute(a, b) =
  match (BigInt.of_string(a), BigInt.of_string(b)) with
  | ( { some = i_a }, { some = i_b } ) ->
    i_add = BigInt.add(i_a, i_b)
    BigInt.to_string(i_add)
  | ( { none }, _) -> "\"{a}\" is not a valid big int"
  | _ -> "\"{b}\" is not a valid big int"

/**
 * Action when the user will click on the 'add' button.
 * get the values of the 2 inputs, and store the result
 * in the result box
**/
action() =
  a = Dom.get_value(#input_a)
  b = Dom.get_value(#input_b)
  Dom.transform([ #result <- compute(a, b) ])
```

```
button(id, message) =
  <a id={id:string}
     class="button"
     ref="#"
     onclick={_->action()}>{message:string}
  </a>

page() =
  <>
  <h1>Big int Binding</h1>
  <h2>Arguments</h2>
  <input style="width:100%" id="input_a"/><br/>
  <input style="width:100%" id="input_b"/>
  <h2>Addition (on the server, using the external implementation)</h2>
  <div id="result" />
  <br/>
  {button("addition", "add")}<br/>
  </>

server = one_page_server("Big int Binding", page)
```

B.3.4 Opa types

Using Opa types in plugin requires a good knowledge of Opa, and can be considered as advanced use. The purpose of Opa types is to create or to wander Opa structures in the external implementation.

This is needed when you want to return or to take in argument complex values.

Tip
For people familiar with C/Ocaml bindings, the manipulation of opa values using the server_lib or the client_lib correspond to the macro CAMLparam, CAMLreturn, Store_field, etc.

B.3.4.1 ClientLib

Coming soon...

B.3.4.2 ServerLib

Coming soon...

B.3.4.3 Complete example using Opa types manipulation

This example shows how to use the opa-type construction from OCaml using the ServerLib. We take the opportunity to bind *Opa* with the NDBM database.

The example is a tiny application for submitting triplets in a persistent external database, finding back the information, and guessing the age of the Captain.

For compiling the example you need: -ndbm dev library; -ndbm OCaml binding.

Caution
This example is of course heretic because *Opa* has its own database. The purpose is to take benefits of a small example to show that you can really extend *Opa* as much as you want using *OCaml* bindings (ndbm is implemented originally in C).

```
user@computer:~/$ ls
dbm.ml  dbm.opa  Makefile
```

Makefile

```
### if Opa has been compiled with Dbm support, nothing to add (comment these lines)
### if Opa compiler is without Dbm support, uncomment them
### dbm is in a subdirectory of ocaml libs
INCLUDE=-I +dbm
LINK=dbm.cmxa


all : plugin opa

plugin:
        $(OPA_PLUGIN_BUILDER) $(INCLUDE) -o dbm dbm.ml

opa:
        $(OPA) $(INCLUDE) $(LINK) dbm.opp dbm.opa

clean:
        rm -rf _build _tracks dbm.opp dbm.exe *.log
```

dbm.ml

```
(**
   1)
   Minimal Opa Binding for Dbm
*)

##extern-type dbm = Dbm.t

(**
   Open a dbm database given its name.
   Return None in case of an error
*)
##register open_db : string -> option(dbm)
let open_db name =
  let flags = [ Dbm.Dbm_rdwr ; Dbm.Dbm_create ] in
  let permission = 0o644 in
  try
    Some (Dbm.opendbm name flags permission)
  with
  | _ -> None

(**
   Close the database
*)
##register close_db \ ` Dbm.close` : dbm -> void

(**
   Find
   Option for opa
*)
##register find : dbm, string -> option(string)
let find dbm key =
  try
    Some (Dbm.find dbm key)
  with
  | Not_found -> None
```

```
(**
   Add or replace
*)
##register add \ `Dbm.replace` : dbm, string, string -> void

(**
   2)
   Specialization of the Dbm database
   for a tiny 3 fields database
*)

(**
   We will store in the database a list of people, indexed by their name
   The type [person] is defined in opa, and we will manipulate it there
   using an [opa-type] construction, with the [ServerLib]
*)
##opa-type person

(**
   We will build a representation in Ocaml
*)
type person = {
  name : string ;
  age : int ;
  email : string ;
}

##extern-type ocaml_person = person

(**
   Implementing a traduction between opa and ocaml structures
*)

(**
   We should define the field of the structure
*)
let field_name = ServerLib.static_field_of_name "name"
let field_age = ServerLib.static_field_of_name "age"
let field_email = ServerLib.static_field_of_name "email"

(**
   From Ocaml to Opa
*)
##register opa_person_of_ocaml_person : ocaml_person -> opa[person]
let opa_person_of_ocaml_person ocaml_person =
  let { name ; age ; email } = ocaml_person in
  let opa_person = List.fold_left
    (fun acc (field, value) ->
       ServerLib.add_field acc field value)
    ServerLib.empty_record_constructor
    [
      field_name, Obj.repr (ServerLib.wrap_string name) ;
      field_age, Obj.repr (ServerLib.wrap_int age) ;
      field_email, Obj.repr (ServerLib.wrap_string email) ;
    ]
  in
  wrap_opa_person (ServerLib.make_record opa_person)

(**
   From Opa to Ocaml
*)
##register ocaml_person_of_opa_person : opa[person] -> ocaml_person
```

```
let ocaml_person_of_opa_person opa_person =
  let record = unwrap_opa_person opa_person in
  let name = ServerLib.unsafe_dot record field_name in
  let age = ServerLib.unsafe_dot record field_age in
  let email = ServerLib.unsafe_dot record field_email in
  {
    name = ServerLib.unwrap_string name ;
    age = ServerLib.unwrap_int age ;
    email = ServerLib.unwrap_string email ;
  }

(**
   Add a person in a database, printing some logs on the server side
   For the simplicity of this tutorial, we use a marshaling of the
   ocaml structure.
*)
##register add_person : dbm, opa[person] -> void
let add_person dbm person =
  let ocaml_person = ocaml_person_of_opa_person person in
  Printf.eprintf "adding %S/%d/%S in the database\n%!"
    ocaml_person.name
    ocaml_person.age
    ocaml_person.email
  ;
  Dbm.replace dbm ocaml_person.name (Marshal.to_string ocaml_person [])

(**
   Find a person in a database
*)
##register find_person : dbm, string -> option(opa[person])
let find_person dbm name =
  match find dbm name with
  | None -> None
  | Some ocaml_person ->
      let ocaml_person = Marshal.from_string ocaml_person 0 in
      Some (opa_person_of_ocaml_person ocaml_person)
```

dbm.opa

```
/**
 * Simple page using dbm
**/

import stdlib.system

type dbm = external

type person = {
  name : string ;
  age : int ;
  email : string ;
}

/**
 * We open the database when the server is initializing,
 * and close it at the end of the execution, e.g. after a SIGINT
**/
dbm : dbm =
  name = "people.dbm"
  dbm =
    match %%dbm.open_db%%(name) with
    | { some = dbm } -> dbm
```

```
    | { none } -> @fail("cannot open dbm database at {name}")
  do System.at_exit(-> %%dbm.close_db%%(dbm))
  dbm

result(string) = Dom.transform([ #result <- string ])

/**
 * Find an entry in the ndbm from the contains of the Name box
**/
@publish action_find() =
  do Dom.clear_value(#age)
  do Dom.clear_value(#email)
  name = Dom.get_value(#name)
  match %%dbm.find_person%%(dbm, name) with
  | { some = person } ->
    do Dom.set_value(#age, string_of_int(person.age))
    do Dom.set_value(#email, person.email)
    void
  | { none } -> result("\"{name}\" Not-found")

/**
 * Submit the truplet in the ndbm
**/
@publish action_submit() =
  name = Dom.get_value(#name)
  age = Parser.int(Dom.get_value(#age)) ? 0
  email = Dom.get_value(#email)
  person = ~{ name age email }
  do %%dbm.add_person%%(dbm, person)
  do Dom.clear_value(#name)
  do Dom.clear_value(#age)
  do Dom.clear_value(#email)
  do result("Done")
  void

/**
 * Guess the age of the captain
**/

the_captain = "The Captain"
the_question = "What is the age of \"{the_captain}\" ?"

@publish action_captain() =
  match %%dbm.find_person%%(dbm, the_captain) with
  | { some = person } ->
    result("The captain is {person.age} years old")
  | { none } ->
    result("I don't know")

page() =
  <>
  <h1>Dbm Binding</h1>
  <h2>New entry</h2>
  <table border="1">
  <tr>
    <th>Name</th>
    <th>Age</th>
    <th>Email</th>
  </tr>
  <tr>
    <td><input id="name"/></td>
    <td><input id="age"/></td>
```

```
    <td><input id="email"/></td>
  </tr>
  </table>
  <a class="button" ref="#" onclick={_->action_find()}>Find</a>
  <a class="button" ref="#" onclick={_->action_submit()}>Submit</a>
  <br/>
  <a class="button" ref="#" onclick={_->action_captain()}>{the_question}</a>
  <br/>
  <div id="result" />
  </>

server = one_page_server("Dbm Binding", page)
```

Exercise Add a primitive which return a opa-list of entries, and add a button to show all entries of the database.

Appendix C

The type system

About this section

This section details the type system of Opa, including:

- definition of types
- polymorphism and rank 2 polymorphism
- interactions with packages and modules
- type error messages
- type best practices

C.1 Opa type system overview

Strong typing Opa is a **statically** and **strongly** typechecked language. This means that the type of an expression is computed once and for all at compile-time, an expression has one unique type and this type can't be used as another one. In Opa, any expression evaluates into a value and functions are first class values. This means that, like integers, booleans, strings in other languages, functions in Opa can be passed as arguments, returned as results, stored in data-types, and so on.

Type inference Opa compiler offers **type inference**, i.e. it automatically determines the type of expressions from the context, without the user needing to write these types. Hence, a definition like

```
x = 5
```

will lead `x` to have the type `int` because the integer literal `5` is known to be of type `int`, this type representing the type of integer values.

As a bit more complex example, the function

```
chose(cond, if_true, otherwise) = if cond then if_true else otherwise
```

will be found having the type

```
(bool, int, int) -> int
```

where `bool` represents the type or boolean values and `->` is the function taking arguments of the types stated on the left side and returning a result of the type stated on the right side of the arrow symbol.

Polymorphism The type system of Opa features **polymorphism**, i.e. types in which some parts are not constrained and can be of any type. For instance, the identity function:

```
identity(x) = x
```

doesn't set any constraint on it argument's type: it can safely be applied to any kind of value and will return this kind of value. Its type will then be `'a -> 'a` where `'a` is a **type variable**, i.e. "*any type will do for 'a*". Using `identity` with an integer as argument will then instantiate `'a` with `int`. Using `identity` with a boolean will then instantiate `'a` with `bool`.

Records and sums In addition to usual data-types (integers, booleans, string, floating point numbers), Opa provides two cornerstone structures: **records** and **sums**. A **record** can be seen as a `struct` in C, i.e. a non-ordered sequence of fields packed into one global entity. However, conversely to C these records can be either "*closed*" or "*opened*": the list of fields can be fixed or can contain "at least" the ones stated in the record type. A **sum** is a set of possible "cases", each case being a record. Hence, a sum allows to express the fact that a data-structure can be of different shapes. This can be compared to `union` in C, with the difference that the effective shape of the data is directly embedded inside its representation.

Type abstraction In the Opa system, **type abstraction**, i.e. ability to hide the actual structure of a type is provided thanks to the notions of **private** and **abstract** types. The first ones are types that only have an existence in the package where they are defined; they are not visible outside this package. The second ones are transparently visible (i.e. their structure can be manipulated) only in the package where they are defined; outside this package, they become opaque, i.e. are still visible but their internal structure is not visible anymore.

Modularity Opa provides a certain notion of **module**, i.e. packages containing value definitions via a particular form of records. Such records are syntactically marked as being modules and will get a particular type called **type forall**. Under some restrictions due to typechecking decidability, it is possible to use these modules like regular record, hence passing them through functions or returning them as result.

C.2 Type inference: a "slight" restriction

Type inference is not complete for some parts of Opa features and a few type annotations are required from the user. This is mostly due to the expressiveness of the language that provides very flexible features at the price of the non-existence of a principal type. Existence of a principal type means that there is always one "*better*" type, i.e. all the other types that could be chosen for an expression would reduce the contexts where this expression can be used. In such a case, the user will have to indicate himself the type he wishes using a type annotation to enforce the expression to be of this type. This construct is called a **coercion**. On the other side, coercion by type annotations may also be required when the typechecker infers a cyclic type. In effect, such types are internally used by the typechecker but are not currently allowed outside in the language.

C.3 Opa type algebra

As previously stated, expressions evaluate to values of certain types. The type algebra describes the shape of these types. The present section doesn't explains how types are synthesized from expressions even if some examples are given. We rather address here an intuitive presentation of the types.

C.3.1 Constant types

Natively, Opa provides three basic types:

- `int` : The type of signed integer values (e.g. `0`, `-4`).
- `string` : The type of characters strings (e.g. `"this is a string"`).
- `float` : The type of signed floating point values (e.g. `0.489`, `-5.94`).

C.3.2 Type variable

As stated in introduction, type variables are used to represent types with not particular constraint, i.e. types we don't need to know about or we don't know yet their effective internal structure. Type variables are written by a leading quote followed by regular identifier, for instance: `'a`, `'myvar5`.

Type variables act in two particular parts of the type system.

- During type inference, they are used by the typechecker to temporarily give types to expressions that at a certain point do not exhibit the need to have a particular type.
- At the end of type inference of a definition, type variables that still remain unconstrained are "*generalized*". This means that the typechecker discovered that there is no need to know more about the structure of the types they represent and hence decides to make the type polymorphic: any type will be suitable in place of these variables. Hence, in the type of a definition (not expression), all finally appearing type variables are implicitly quantified by a "*for all*" qualifier, telling that the definition having this type has all the possible **type schemes** where type variables can be instantiated (replaced) by any type. In other words, a type scheme is a "*model*", a "*pattern*" of types, describing a "*family*" of possible types.
  This last point raises an important concept:
  - **Expressions** have a **type**.
  - **Definitions** have a **type scheme**.
    In other words:
  - In an expression a type variable is just an **unknown** that may one day be known equal to some type.
  - In a definition a type variable is a **universally quantified type**. Each instance, usage of the identifier bound by this definition is free to replace these variables by types it needs in its context.

C.3.3 Function type

A function is characterized by the type of its arguments and the type of its result. The type constructor for a function is the "*arrow*" symbol: `->`. On the left part the types of the arguments are listed, comma-separated if there are several arguments. On the right part is the type of the result of the function.

Examples:

- `-> int` : Function taking no argument and returning an integer.
- `int -> bool` : Function taking one argument of type integer and returning a boolean.
- `int, string -> string` : Function taking two arguments, the first one being of type integer, the second of type characters string and returning a characters string.
- `int, (int -> bool), int -> bool` : Function taking three arguments, the first and third ones being of type integer, the second one being a function taking an integer and returning a boolean. Finally, the function returns a boolean. Note the presence of parentheses enclosing the function type of the second argument.

C.3.4 Named type

Types can be inductively combined to form complex data-types and bound to names thanks to **type definitions**. Hence, the following extract defines a new type name `natural`, whose structure is `int`.

```
type natural = int
```

Once this type is defined, we may use type name `natural` to denote this new type.

Types can have parameters. For instance, the type of pairs with their two components being of a same type can be defined as:

```
type same_comp_pair_ty('a) = ('a, 'a)
```

A pair denoted by the type `same_comp_pair_ty(float)` will have its two components carrying floating point numbers.

The type of pairs with components of any types can be defined as:

```
type same_comp_pair_ty('a,'b) = ('b, 'a)
```

Note that the order of the type variables `'a` and `'b` is different in the quantification (i.e. left-part of the definition, on the left of the = sign) and in the body of the type definition. As a result, this means that pairs having type `same_comp_pair_ty(int, bool)` will carry integers in their second component and booleans in their first component.

Finally the type of relational operators (i.e. functions taking two arguments of the same type and returning a boolean value) can be defined as:

```
type relational_ty('a) = 'a, 'a -> bool
```

Following this definition, the type of relational operators testing integers is `relational_ty(int)`.

Hence, a named type is an identifier followed by as many types comma-separated and enclosed between parentheses as the corresponding type definition has parameters.

See also the section dealing with type definitions.

C.3.5 Sum type

Sums are a sequence of cases, each case being a record type.

A record is a sequence of typed fields. In a record, accessing a fields will yield a value of the type of this field. Sums as record can be either **closed** of **opened**. A **closed** record type (respectively sum) enumerates exactly the fields it contains (respectively cases for a sum). An **opened** record type (respectively sum) enumerates the fields (respectively cases for a sum) it may contain at minimum, and allows to have more if needed. In case of opened record or sum, the type ends by a trailing ellipsis, otherwise it is silently closed.

An **opened** sum ends by a **column variable**, i.e. a kind of type variable that can be instantiated by another sum. In other words, this column variable allows the sum to gain more sum cases.

An **opened** record ends by a **row variable**, i.e. a kind of type variable that can be instantiated by another record (also called "*row*"). In other words, this row variable allows the record to gain more fields.

In error messages (where they need to be clearly named), column variables (resp. row variables) are denoted by `'c.a`, `'c.b`, etc (resp. `'r.a`, `'r.b`, etc). In other cases, when types are printed and the identity of such variables doesn't matter, they are both written "...".

Here follow some examples of correct record types :

- `{ }` : Empty record type. It contains no field.
- `{ x : int ; y : float }` : Closed record containing exactly the two fields `x` of type `int` and `'y` of type `float`.
- `{ x : int ; ... }` or `{ x : int ; ... }` : Opened record type containing at least a field `'x` of type `int` and possibly any other fields of any type.

Sums are a form of sequence of records. They can be non-disjoint, meaning that some records forming cases may overlap.

Important
A strong restriction imposes that an **opened sum** must not host **any** case that is an opened record and any (**closed or opened**) **sum** must not host **several** cases that are opened records. Moreover, if several cases (i.e. records) of a sum share a same label, this label must have the same type in each cases. Finally, if a sum has several cases, none of them must be opened.

Here follow some correct sum types examples:

- `{ a : int } / { b : float }` : Closed sum containing two cases, one being a record having exactly one field `a` of type `int`, the second begin a record containing a record having exactly a field `b` of type `float`.
- `{ a : int } / ...` : Opened sum containing at least one case being a record having exactly a field `a` of type `int` and possibly any other cases.
- `{ a : int } / { a : int ; b : float }` : Closed sum containing exactly two cases, one being a record containing exactly one field `a` of type `int`, the second being a record containing exactly two fields, `a` of type `int` and `b` of type `float`. We see in this example that fields can be common to several cases as long they keep the same type. For example, having `a` of type `bool` in one of the cases would be ill-typed.
- `{ a : int } / { a : int ; b : float } / ...` : Opened sum type similar to the previous example. It may contains additional other cases compared to the previous example.
- `{ a : int }` : Closed sum with only one case. It can be hence seen as a "record" (i.e. like a `struct` a la C) exactly containing one field `a` of type `int`.
- `{ a : int ; ... }` : Closed sum with only one case. This case is opened hence can be seen as a record containing at least one field `a` of type `int`. Any other record containing additional fields is compatible with this type.

C.3.6 Type forall

We saw in section Type variable the difference between **types** and **type schemes**. Variables present in type schemes are universally quantified, hence a type scheme represent a family of types. A **type forall** embeds a type scheme inside a type and represents the type of a module field. Such a type is written by prefixing its body by the keyword `forall` followed by the list of polymorphic type variables of the type between parentheses followed by a "`.`" (dot) then the body of the type. Type-forall are in fact a way to introduce a flavor of rank-2 polymorphism in Opa type system.

Rank-2 polymorphism is a system in which a forall quantifier may not appear to the left of more than 2 arrows (when the type is drawn as a tree). To illustrate this concept, consider the function:

```
f(x) = x
_ = (f (true), f(2))
```

variables bound with a definition (sometimes called "*let-definition*" may be polymorphic, as in `f` above. That's the reason why `f` can be applied to two different types, `bool` and `int`. However, in a rank-1 polymorphism system, function arguments may not be:

```
g(f) = (f(true), f(2))
```

results in an error:

```
Error
Function was found of type bool -> 'a but application expects it to be of
type int -> 'b.
Types int and { false } / { true } are not compatible
```

In a rank-2 polymorphism system, we can give `g` a signature like `(forall a. a -> a) -> (bool, int)`. This is called a rank 2 type, because a function argument is polymorphic, as indicated by the forall quantifier. Now the function `g` may be applied to expression whose generalized type is at least as general as that declared.

Modules are particular records in which each field type gets generalized. This way, a module can be seen as a package of definitions grouped in the module rather than laying at toplevel.

When a module field definition appears to be polymorphic, its type is turned into a type forall where all generalizable variables are universally quantified. The type of the module is hence a **closed** record type whose fields names are the fields definitions names with their related type turned into a type forall if they are polymorphic. Hence, types forall are closely related to module fields types and can only be created via module expressions.

The following program clearly show the difference between the type of `r` which is a regular record and the one a `m` which is a record containing types forall for fields having a polymorphic type.

```
r = {
  id(x) = x
  pair(x) = (x, x)
  one = 1
}
r : {id : 'v0 -> 'v0; pair : 'v1 -> tuple_2('v1, 'v1); one : int} / ... =
 { id = <fun> ; one = 1 ; pair = <fun> }

m = {{
  id(x) = x
  pair(x) = (x, x)
  one = 1
}}
m : {id : forall('v0). 'v0 -> 'v0;
      pair : forall('v1). 'v1 -> tuple_2('v1, 'v1);
      one : int} =
 { id = <fun> ; one = 1 ; pair = <fun> }
```

In the record type of `r`, no type forall appear even if the type scheme of `r` itself is polymorphic (`'v0` and `'v1` are generalized). In the record type of `m`, since it is a module, all fields having a polymorphic type get generalized and turned into forall types. One can notice that `one` being not polymorphic, it is not turned into a forall type.

Important
Types forall are explicitly created by the developer when he uses a module construct: there is hence no automatic type forall creation. Conversely, when a type forall appears in an expression, it is automatically instantiated, hence leading to a type in which generalized variables are replaced by fresh variables and universal quantification disappears (in other words, the type forall becomes a type "non"-forall again).

C.4 Opa type algebra formal grammar

For sake of completeness, we give here a presentation of the type algebra like usually described in theoretical approaches. This section is mostly dedicated to advanced users or people familiar with type systems design. A more complete description of this theoretical presentation is available in the technical and architecture documents of Opa.

Type τ ::= `int` Integer constant type
| `bool` Boolean constant type
| `string` String constant type
| `float` Floating point numbers type
| `void` Trivial (i.e. having 1 unique constructor) constant type
| α Type variable
| t (τ_i) Named type constructor (possibly parameterized)
| τ -> τ Function type
| κ Sum type
| $\forall\ \alpha_i, \rho_j, \delta_k\ .\ \tau$ Type forall

Row φ ::= • Closed row
| ρ Row variable
| l : τ ; φ Field sequence

Sum κ ::= • Closed column
| ζ Column variable
| { φ } / κ Case (record) sequence

C.5 Typing a program

Now that the form of Opa's types has been presented, we will address how Opa programs are typechecked. The aim of this section is to understand the type to expect for each construct of the language. This presentation is split in two parts, one dealing with type definitions and one dealing with expressions.

C.5.1 Typing type definitions

Type definitions allow to create new data-structure from previously existing ones or allow to alias, i.e. provide a name as a shortcut for cumbersome type expressions. Both usages follow the same syntax and rules.

C.5.1.1 Structure of a type definition

A type definition is introduced by:

```
[directive] type [directive] ident[(parameters,...)] = type expression
```

where *ident* is the name of the created type, *parameters* is a comma-separated enumeration between parentheses of the type variables parameterizing the definition (or nothing if the definition is not parameterized), and *type expression* is the body of the defined type. We will see directives in the next section.

Once defined, the new type name will be compatible with itself and any type expression compatible with the body of its definition. Hence, following the definition `type t = int`, the type `t` can be used anywhere an `int` is expected and reciprocally.

C.5.1.2 Directives in type definitions

Directives allow to alter the visibility of a type definition to ensure abstraction. Two directives are available: `@private` and `@abstract`.

- `@private` makes the type definition only visible in the package where it it defined. Outside this package, the type is no more visible : it doesn't exist for other packages. Such types are typically "internal tools" of a package and serve only to implement functionalities without appearing in the interface of the package. As a consequence, they must not escape outside the package, i.e. must not appear in the signatures of values visible outside the package. To make this possible, values manipulating such types must also be declared private, using the `@private` directive also available for value definitions. If a private type appears in the signature of an exported value, an error is raised, advising to make this value private, like in the following example:

```
package private_ty_escape2
@private type prv_string_ty = string
vv = "" : prv_string_ty
Error
Description:
Definition of vv is not private but its type contains a type private to the
package. Type prv_string_ty must not escape its scope.
Hint:
  Add a @private directive on this definition.
```

- `@abstract` makes the type definition transparently visible inside the package where it is defined, i.e. its internal structure can be accessed and manipulated, but outside this package it turns opaque: the type is still visible (i.e. it still exists) but its internal structure is not visible anymore. Such types are typically data-structures implemented in a package, provided to other packages that must not directly manipulate the effective implementation of the data-structure and instead use the API provided by the package implementing the data-structure. This is a kind of abstract data type (ADT), which permits to preserve consistency and internal invariants required by the implementation of a data-type. The basic idea is that the definition package "knows" how to safely manage the type, provides a safe API to the outer "world", hence one gets sure that nobody will come and "hack by hand", hence break, invariants required by the data-structure to keep consistent. Outside its definition module, an abstract type loses its internal representation, hence becomes compatible only with itself.

```
package p1
@abstract type u = int
to_u(x : int) : u = x
from_u(x : u) = x : int
// Legal since we are in the defining package: we see u's internals.
_ = 1 + (1 : u)
```

```
package p2
import p1
// Obtain a value of type u from the integer 0.
v = to_u(0)
_ = 1 + v
```

Because the last line of code of package `p2` tries to rely on `u`'s internal structure although it is not the package defining it, `u` being turned abstract can't be compatible with `int` anymore, hence leading to the error:

```
Error
Function was found of type 'a, 'a -> 'a but application expects it to be of
type int, u -> 'b.
Types int and u are not compatible
```

C.5.1.3 Parameters in type definitions

Parameters allow to define types polymorphically, in order to instantiate them later by providing effective type arguments to the type name in future type definitions or type expressions. Let's examine the following sample code.

```
type fun_t('a, 'b) = 'a -> 'b
type int_string_fun_t = fun_t(int, string)

twice_general(f , i) = f(i) + f(i)
twice_general : ('v0 -> int), 'v0 -> int = <fun>

twice(f : fun_t(int, int), i) = f(i) + f(i)
twice : fun_t(int, int), int -> int = <fun>
```

`fun_t` is defined as the type of functions taking one argument an returning a value, both of them not being fixed in the definition. Hence the type `fun_t` is "*partially defined*" and will need to be instantiated with effective arguments in further usages.

Next comes the definition of `int_string_fun_t` using `fun_t` and specializing this latter with `int` for both the types of the argument and returned value of the function. By this definition, we used the general `fun_ty` to create a more specialized type definition.

Let's now consider the definition of `twice_general` taking a function, a parametric value and adding twice the result of applying this function to this value. The type of + (internally `int, int -> int`) enforces the result of `f` to be `int` but no constraint applies on the type of the argument of `f`. Hence `f` has type `'v0 -> int`, and by consequence `twice_general` also has a parameterized type `('v0 -> int), 'v0 -> int`.

Using the type definition `int_string_fun_t`, we can define a function similar to `twice_general`, i.e. with the same body but with a more restricted type: `twice`. Hence, by adding a type coercion on the parameter `f` of `twice`, we can enforce it to be of type `fun_t(int, int)` hence forcing the parameter of `f` to be `int` instead of leaving it free like in `twice_general`.

Note
Note that both definitions of `fun_t` and `int_string_fun_t` do not really create "new" types. They are aliases, "*shortcuts*" for a type expression. Instead of using these 2 defined type names, we could directly use their body but this would be less handy. Creation of really new types is achieved by defining sums and records as described below.

C.5.1.4 Using type expressions in a type definition body

The basic type expressions that can be used allow to represent the types described in the Opa type algebra.

- **Basic constant named types** : `int`, `bool`, `string`, `float`, `void`.
  `type t = int` defines `t` as an alias on the type of integers `int`.
- **Function type** : `->`
  `type t1 = int -> float` defines `t1` as the type of functions taking an `int` and returning a `float`.
  `type t2 = -> float` defines `t2` as the type of functions taking no argument and returning a `float`.
  `type t3('a, 'b) = 'a -> 'b` defines `t3` as the type of functions taking an argument and returning a value whose types are not fixed and are parameters of the type definition. Hence, `t3(int, float)` specializes these parameters and lead to a type compatible with the above type `t1`.
  `type t4 = int, string -> bool` defines `t4` as the type of functions taking two arguments (of types `int` and `string`) and returning a `bool`.
- **Named type**
  Type definitions bind type expressions to names. So defined type names can be on their turn used to form new type definitions. If a type name represents a parameterized type, it must be instantiated providing effective type arguments according to its arity (i.e. number of required arguments).
  `type t5 = t1 -> t2` defines `t5` as the type of functions taking a parameter of type `t1` and returning a result of type `t2`, both types being those presented in the above paragraph.
  `type t6 = t3(bool, int)` defines `t6` using the previously defined type `t3` and applying it to `bool` and `int`. As a consequence, `t6` is the type of functions taking an argument of type `bool` and returning a value of type `int`.
- **Sum**
  A sum type is an enumeration of "*cases*" separated by a "/" (slash character). Each "*case*" is a "*record*" i.e. an enumeration between braces of labels with their type.
  `type t7 = { A : int } / { B : string ; C : bool }` defines `t7` as a new sum type containing two "*cases*". The first one is a record containing a field `A` of type `int`; The second one is a record containing two fields, `B` of type `string` and `C` of type `bool`. Values of this type can be "*pattern matched*" to discriminate their effective form and access the particular fields of each case.
  Sum type definitions always lead to "*closed*" sum, i.e. the resulting type doesn't ends by a "..." (column variable). This simply means that when defining a type, the definition simply describes all the "*cases*" this type can host.
  As a shortcut, when a field of record has type `void`, its type specification can be ommited. Hence,
  `type color = { Red } / { Green } / { Blue }` defines `color` as a sum with three cases, each of them implicitly of type `void`. This is equivalent to `{ Red : void } / { Green : void } / { Blue : void }`. The idea in such a definition is that the fields of the record are self-sufficient and do not need to carry additional information to be precise enough.
  `type pair('a, 'b) = { first : 'a ; second : 'b }` defines `pair` as a parameterized sum with only one "*case*". This "*case*" is a record containing two fields, one per component of the pair. Since the pair data-structure is naturally polymorphic, this type definition has in effect 2 parameters `'a` and `'b`.

Important
Remember, as already state while presenting Opa type algebra, in the sum types section, that a strong restriction imposes that an **opened sum** must not host **any** case that is an opened record and any (**closed or opened**) **sum** must not host **several** cases that are opened records. Moreover, if several cases (i.e. records) of a sum share a same label, this label must have the same type in each cases. Finally, if a sum has several cases, none of them must be opened.

- **Record**
  A record type is defined like (and compatible with) a sum type with only one "*case*". This way, by default a record type is a closed row in an opened column.
- **Type forall**
  Type-forall being strongly related to modules, they are defined using a syntax close to modules, allowing to avoid the explicit quantification of type variables. In fact, type-forall are not directly written, they are introduced on need through module types.

Any type variable appearing in the type of a field of the definition and not bound outside will be implicitly considered as quantified.
`type t = {{ identity : 'a -> 'a }}` defines `t` as a module type (i.e. a record) in which the field `identity` has the type-forall `forall('a).'a -> 'a`.
Note that variables not bound outside the module type have a scope local to the field's type where they appear. For instance, in `type t = {{ identity : 'a -> 'a ; val : 'a }}`, `'a` represents 2 different variables in `identity` and `val`.
Variables bound outside the module type are not quantified, hence not participating into type-forall creation. For instance in `type t('a) = {{ id : 'a -> 'a ; v : 'b }} -> 'a` the type variable `'a` being a type parameter is bound outside the module type, hence is not quantified. By consequence, the type of the field `identity` is no more a type-forall. Conversely, `'b` being not bound remains quantified and introduces a type-forall. This also shows that type-forall are introduced only when polymorphism exists: there exists no "*trivial*" type-forall, i.e. no type-forall with an empty set of quantified variables.

A few syntactic extensions allow shortcuts to make life easier.

- **Sum sugar**
  Additionally, a syntactic shortcut allows to merge several sum types definitions to create a new definition containing the "*cases*" of all the initial types. In the same idea than when creating a sum type, i.e. separating its different cases by a "/", the names of types to merge are simply separated by a "/".
  ```
  type t = { A } / { B }
  type u = { C }
  type v = { D }
  type w = t / u / v
  _ = { C } : u
  ```
  _do_ : u = { C = void ; }
  In the above sample code, `w` finally stands for the type definition `{ A } / { B } / { C } / { D }` and expression `{ C }` can really be coerced into `w` although it is initially a value of type `u`. You may note the `_ = expr` construct which allows evaluating the expression *expr* then throw its result value. Since this value is not bound to any identifier, Opa names it `_do_` in its feedback messages.

- **Tuples**
  Tuples are generalization of pairs. A pair is a data-structure with 2 components: a tuple has an arbitrary number of such components. Two tuples types are compatible if their components are compatible 2 by 2. This especially means that 2 compatible tuples types must at least have the same number of components. Syntactic sugar allows to write tuples types as comma-separated sequence of types enclosed between parentheses.
  `type t10 = (int, bool, int, int)` defines `t10` as a tuple with 4 components, all being of type `int` except the second one.
  `type t11('a) = ('a, 'a, 'a)` defines `t11` as a tuple with 3 components all being of the same type, parameter of the type definition. Using `t11` in a type expression with `bool` as effective argument will hence represent the type of a tuple with 3 components all being booleans.

Note
Opa standard library already proposes numerous data-types like lists, maps, sets, etc which may be directly used in developments, avoiding the need to implement again the related common and widespread algorithms.

C.5.2 Typing expressions

In this section, we will review each Opa construct and explain which way its type is computed. In some cases, we will see that inference is not complete and type annotations may be added by the developper to help the typechecker in its job.

C.5.2.1 Constant expressions

Constant expressions are given their natively defined type:

- Integer values: `int`
- Boolean values (`{ true }` and `{ false }`): `bool`
- Characters strings literals: `string`
- Signed floating point values: `float`
- The trivial type value (`{}`): `void`

Examples:

```
i = 5
i : int = 5
f = 3.14159
f : float = 3.14159
s = "foo"
s : string = "foo"
v = {}
v : {} / ... = void
b = true
b : bool = { true = void ;  }
```

First, notice the apparition of a construct *ident* = ... we will detail deeper later. For the intuition, let's simply say it **binds** ("*attaches*" the result of its right-side expression to the name *ident*). This allows to name values. Be aware that this is different from the notion of **assignment** "à-la C" which permits to modify in place the value stored in the memory cell touched by the assignment. The difference between a **binding** and an **assignment** refers to the notion of mutability, i.e. the possibility to alter the value represented by the identifier. A binding links an identifier once for all to a value. Some kind of values are mutable, this is the way assignments are obtained aside bindings.

Note in the above sample code that the void and boolean values have been typed as sums, not as named types `void` and `bool`. This is due to the fact that type inference is structural: named types are introduced by type annotations. Types `bool` and `void` are just abbreviations on the structural types `{}` and `{ true = void } / { false = void }`. If no explicit type annotation imposes to see these void and boolean values as `void` and `bool`, the structural types induced by their structures remain the primarily ones inferred.

C.5.2.2 Functions

A function may have arguments but always return a value. Hence, defining a function requires to define its possible arguments and write its body, i.e. the expression building the returned value. The form of a function expression is:

```
[parameters,...] -> expr
```

Parameters of the function are bound by the definition, hence are only available in the function's body. Obviously, to typecheck the body of the function, we will need the types of its arguments. Type schemes bound to identifiers (function parameters or identifiers introduced by the identifier binding construct) are recorded in an internal structure we call here a "*typing environment*" acting as the memory of "*already seen identifiers*'" types.

Typechecking a function expression is performed in 3 steps:

1. First, each parameter is given a temporary "*unknown*" type: a trivial type scheme (i.e. with no generalized variables) whose body is a type variable. **This especially means that a function parameter can't be used polymorphically inside the body of the function.**

2. The body *expr* is then typechecked in the typing *environment* extended by the function's parameters, hence leading to the type τ of the returned value.

3. Finally, an "*arrow*" type is built, grouping the types of the parameters (that were possibly instantiated because of type constraints found while typechecking the function's body) on the left-side of the "*arrow*" and setting the returned type on the right-side, finally giving a type of the form $\tau_1, \ldots, \tau_n$ -> τ.

Examples:

```
_ = x -> x + 1
_do_ : int -> int = <fun>
```

First, notice the `_ = ...` notation that allows to throw the result of the expression laying on its right-side. This is a particular form of identifier binding, a construct described in detail later. For the moment, the intuition to keep is that it binds the result of the expression to "*nothing*".

In this sample code, `x` is inserted in the typing environment with a type variable, say `'a` (what we previously called an "*unknown*" type). The body of the function is then typechecked. Even is we didn't present yet the typechecking of function application, driven by the intuition, the operator + expects two integers and returns an integer. Hence, to be correctly typed, the parameter `x` of the function must be of type `int` i.e. the type variable `'a` previously introduced for `x` gets instantiated by `int`. Now, `1` also has type `int`, so the operator + is used in a consistent typing context, returning a value of type `int` which is the global type of the function's body. Finally, the whole function type is built having `'a` now being `int` on the left-side of the `->` and `int` on its right-side, giving `int -> int`.

```
_ = x, y -> x + 1
_do_ : int, 'v0 -> int = <fun>
```

This second example shows a function with 2 arguments. As previously, each of them is entered in the typing environment with a type variable before typechecking the body of the function. This body is the same than in the previous example, hence it doesn't use the `y` parameter of the function. As a consequence, no constraint applies on `y` and its type remains a variable. As a final result, grouping the types of the arguments on the left-side of the `->` and the result type on its right, we obtain the final type `int, 'v0 -> int`.

```
_ = -> "constant"
_do_ : -> string = <fun>
```

In this last code snippet, since the function doesn't have any argument, its body is typechecked in the unmodified typing environment. The string literal `"constant"` representing the body of the function had type `string` which is the type of its returned value. Since there is no argument, there is no type to set on the left-side of the final `->` type, only the result type will appear on its right-side, leading to the function type `-> string`.

C.5.2.3 Application

Application is the construct allowing to "*use*" a function by providing it some effective arguments. In Opa, an application expression is denoted by a functional expression on the left-side and the enumeration comma-separated and between parentheses of the expressions being effective arguments to pass to the function:

```
expr1([expr2,...])
```

If the function requires no argument, then the enumeration between the parentheses is empty, however, the parentheses still remain required. Note **that there must not be any blank between the functional expression and the opening parenthesis**. Typechecking an application is performed in 4 steps:

1. First arguments expressions $expr_2, \ldots$ are typechecked, leading to as many types $\tau_2, \ldots$.
2. Next, the functional expression $expr_1$ is typechecked. It is then expected to lead to a functional type with as many arguments' types as there are effective arguments and returning a value of a certain type: τ. Hence, the expected type must have the form $\tau'_2, \ldots \rightarrow \tau$.
3. Each effective argument's type τ_i is checked against the corresponding τ'_i of the type inferred for the function to ensure they are compatible.
4. Finally, the type of the application is the result type of the function, i.e. τ

Examples:

```
f(x, y) = x + y + 1
f : int, int -> int = <fun>
_ = f(2, 4)
_do_ : int = 7
```

To begin, simply ignore the typechecking of the function `f`, we will have a look at it later. Simply accept it has type `int, int -> int` and let's examine the second expression. The arguments expressions `2` and `4` have type `int`. We then typecheck the functional expression which is the identifier `f`. Even if we don't have described typechecking of identifiers yet, we already introduced the notion of "*typing environment*" keeping trace of types of identifiers (see Functions). Looking-up in this environment we find that `f` has type `int, int -> int`. Hence, we see that the types of the effective arguments provided in the application are compatible with the expected types arguments of `f` (i.e. `int`). Hence the result type is the one on the right-side of `f`'s type: `int`.

To go further, we can study how `f` is typechecked since its body contains the application of the operator +. Like previously described for functions, `x` and `y` are entered in the typing environment both with temporary type variables, say `'a` and `'b`. Then the body of the function which is an application must be typechecked. We then first type the arguments, i.e. `x` and `y` that have types `'a` and `'b`. Assuming that + has the obvious type `int, int -> int` we must now ensure that the types of the effective arguments are compatible with the type of +. This means that `'a` must be `int` and same thing for `'b`: the two type variables get then instantiated, i.e. made "*equal*" to `int`. The result type of the application is then the result type of +, i.e. `int`. We iterate exactly the same process to typecheck the addition with the constant `1` which finally gives type `int`. Hence, `int` is the type of `f`'s body, so the returned type of `f`. As a consequence, `f` is inferred the final type `int, int ->int`.

C.5.2.4 Identifier binding

The construct

```
ident = expr1
expr_2
```

allows to evaluate $expr_1$, bind its value to *ident* and evaluate $expr\_2$ in which *ident* possibly appears and will be replaced by its bound value. Identifier binding introduces polymorphism in Opa: once the type of the expression to bind to the identifier is computed, all the type variables remaining in it with no constraint are considered polymorphic and "*generalized*". As a consequence, in identifier binding expressions, identifiers obtain not a type but a type scheme (see also the presentation of type variables) that can be polymorphic.

Typechecking this construct processes in 3 steps:

1. First, $expr_1$ is typechecked, leading to a type τ_1.
2. Then τ_1 gets generalized: all type variables without any constraint are generalized and universally quantified to lead to a type scheme. This type scheme is then bound to the identifier *ident* in the typing environment.
3. Finally, $expr_2$ is typechecked in the extended typing environment, giving a type τ_2 which is the final type of the whole expression.

Examples:

```
v =
  one = 1
  one + one
v : int = 2
```

The identifier `one` is bound to the value `1` which has type `int`. Then it is used in an addition. The operator + expects 2 integers and returns an integer. Hence, since `one` was found of type `int`, it can safely be used with the addition, hence this latter expression leads to the final type `int`.

By the way, we can note a special identifier binding construct, the toplevel one that binds the identifier `v`. In its toplevel flavor, this construct doesn't have any trailing expression (we previously called $expr\_2$). This allows to bind identifiers at the outermost level of a program, i.e. to give it a scope (a "*life duration*") spanning on "*all the rest of the program*".

```
pair =
  identity(x) = x
  (identity(1), identity("foo"))
pair : tuple_2(int, string) = { f1 = 1 ; f2 = "foo" }
```

The above code snippet introduces 2 features. First, the shortcut for defining a function by directly enumerating its arguments. Instead of writing `f = x, y -> 4`, it is possible to write `f(x, y) = 4`. Second, we make use of the built-in syntactic sugar to represent tuples whose components are enclosed between parentheses and comma-separated.

Hence, the identifier `identity` gets bound to a function type directly returning its argument. Clearly, such a function can take any kind of value and will return this kind of value. The only constraint on the type of this function is that it returns the same type it got for its argument, hence giving the type `'a -> 'a`. This type can then be generalized giving `identity` the polymorphic type scheme ∀ `'a.'a -> 'a`. The function is then invoked twice, once with an `int`, once with a `string`, which is legal since it has a polymorphic type. Hence, the results of these invocations have types `int` and `string` and are grouped in a tuple, `tuple_2(int, string)` which is the final type of this expression.

C.5.2.5 Recursive identifier binding

Similarly to the identifier binding construct, the recursive flavor

```
rec ident1 = expr1
...
rec identn = exprn
expr_n+1
```

allows to bind each $ident_i$ to the value of $expr_i$ but with all the $ident_{...}$ possibly appearing in the bodies, $expr_{...}$, of the bound expressions. This hence allows (mutually) recursive definitions. Intuitively, this implies that to compute the type to bind to $ident_i$, we must compute the type of all the expressions $expr_{...}$ for which we may need to already known the types bound to the $ident_{...}$.

Note that toplevel definitions are automatically reordered, i.e. it is possible to use a function and define it later and if needed, definitions are implicitly made recursive. Hence, the `rec` construct mostly serves to define **local** recursive definitions, i.e. definitions not spanning at toplevel (or in other words, definitions nested in other definitions).

Typechecking this construct processes in 5 steps:

1. First, each $ident_i$ is temporarily given a "*unknown*" type: a trivial type scheme (i.e. with no generalized variables) whose body is a type variable. **This especially means that a recursively defined identifier can't be used polymorphically in its own definition.**
   The typing environment gets extended, binding each $ident_i$ to its related temporary type scheme.
2. Then each $expr_i$ is typechecked in the extended typing environment, leading to a type τ_i.
3. Now the bodies of the bindings are typechecked, each τ_i and the related temporary type scheme bound to $ident_i$ (which have possibly be instantiated by some type constraints) are checked to ensure they are compatible.
4. If so, each τ_i is then generalized and becomes the type scheme bound to $ident_i$ in the typing environment.
5. Finally, $expr_{n+1}$ is typechecked in the extended typing environment, giving a type τ_{n+1} which is the final type of the whole expression.

C.5.2.6 Value identifiers

As we previously introduced (see Functions), the typing environment is a memory keeping trace of types inferred and bound to identifiers (bound by a function definition or an identifier binding construct). More accurately, it contains, not the types, but the **type schemes** bound to identifiers since Opa supports polymorphism.

Typechecking an identifier expression aims to find a type for this identifier and is performed in two steps.

1. First a looking-up is performed in the typing environment to find the type scheme entered for this identifiers. Providing that the identifier was actually defined, it das obviously been typechecked and bound a type scheme in this environment.
2. The obtained type scheme must be transformed into a type. Remember that a type scheme is a "*model*", a "*pattern*" of types, describing a "*family*" of possible types. Generalized (i.e. universally quantified) type variables in the scheme represent "*parts*" that can be instantiated at need by any type expression. In order to allow the identifier to be used with different types in place of the type variables of its scheme, the scheme gets "*specialized*", i.e. the type for this occurrence of the identifier is built by copying the scheme's body, replacing all generalized type variables by fresh (i.e. not appearing anywhere in the program) type variables.

Examples:

```
id(x) = x
id : 'v0 -> 'v0 = <fun>
one = 1
one : int = 1
hello = "Hi"
hello : string = "Hi"
_ = (id(one), id(hello), id(id))
_do_ : tuple_3(int, string, 'v1 -> 'v1)
```

In this sample code, the interesting part deals with the 3 applications used to build the tuple expression. We see that `id` was entered in the typing environment with the polymorphic type scheme ∀ `'a.'a -> 'a`.

Hence when typechecking `id(one)`, the type scheme of `one` is fetched, which is not polymorphic and whose body is `int`. Hence, the occurrence of `one` is given the type `int`. The type scheme of `id` is then fetched and instantiated: the quantified type variable `'a` is replaced by a fresh one, say `'b` leading to type `'b ->'b` (note there is no more ∀ since this is a type, not a type scheme). Application then makes this `'b` instantiated into `int` and leading to the final application type `int` for the first component of the tuple.

The second component of the tuple is processed the same way, the type scheme of `id` being fetched again in the environment and specialized again by replacing `'a` by another fresh type variable, say `'c`. The type scheme of `hello` is fetched and is not polymorphic, leading to type `string`. As a consequence, not surprisingly the type of the second component of the tuple is `string`.

Finally remains the third and last component of the tuple. The type scheme of `id` is fetched twice: one first time for the argument of the application, one second time for the functional part of it. Each occurrence of `id` is then given a type obtained by specialization of the scheme, each one having fresh type variables in place of the ones generalized in `id`'s scheme, leading to types `'d ->'d` for the functional usage of `id` and `'e ->'e` for `id` used as argument. Then typechecking of the application imposes that `'d` and `'e->'e` must be the same type. The result of the application expression being `'d`, from the constraint we have just seen, we obtain that `id(id)` has type `'e->'e`, which is the type of the third component of the tuple.

C.5.2.7 Records and sums

A sum expression consists in the record representing the case of sum, i.e. an enumeration (possibly empty) between braces and semi-separated of the field labels with their related values that represent the sum case:

```
{ [label = expr ; ...] }
```

Such an expression always lead to an **opened** sum type containing a **closed** record. The reasons for closed / opened are the following:

- Since the record enumerates all the fields of the case, this means that this record has no more fields in this expression. For this reason, it must be **closed** because it contains no other fields than thoses explicitly stated in the expession.
- Since the expression represents one case of sum, its type at least contains this case, but possibly some other cases. Hence it must be opened to accept being compatible with these possible future other cases.

Typechecking of a record expression is performed in 3 steps:

1. Each field expression $expr_i$ is typechecked, leading to as many types τ_i.
2. A closed record type is then built, binding each $label_i$ to its related found type τ_i. This record being closed means that it doesn't end by a "*row variable*", hence won't be compatible with record containing other fields than those explicitly specified.
3. Finally, an opened sum type is created, plugging the above record type as one of its case. This sum being opened means that it ends by a "*column variable*", hence means that the type of this expression contains "*at least*" this case, but may be more if needed.

C.5.2.8 Pattern-matching

Pattern-matching is the construct allowing to discriminate by cases on the structure of a value. According to the syntax:

```
match expr with
 | case_1 -> expr_1
 | case_2 -> expr_2
  ...
 | case_n -> expr_n
```

the expression *expr* is matched against each pattern in sequence, stopping on the first one that matches. This fitting between the value and the pattern may induce bindings of pattern variables. The environment is then extended by these bindings and the right-side part expression of the matching case is then evaluated as the result of the whole expression.

As a consequence, the matched value and all the patterns must have a same type. Because any right-side part expression of the cases can be returned, they must also have a same type.

More accurately, typechecking of this construct is performed as follows. First, the matched expression is typechecked, leading to a type τ. Then, each pattern is typechecked leading to types $\tau_1, \ldots \tau_n$ that are checked to be compatible together and with τ. While typechecking patterns, pattern variable types are inferred and reminded for later. At this point, a special operation is performed on catchall patterns ("_", or pattern variables) having a sum type. In effect, a catchall pattern absorbs "*any other value not already matched by previous patterns*". In the case of sum types, this means that the column type representing this sum can have "*any other cases*", hence is opened. For this reason, sum types appearing in the type of a catchall pattern get recursively opened (i.e. opening is done in depth). Now the type of the matched expression and patterns is computed, each right-side part expression of cases is typechecked in the environment extended by bindings of its related pattern variables (previously recorded). As a result, each branch gets a type $\tau'_1, \ldots, \tau'_n$ that are checked to be compatible. The resulting type of the whole expression is then the type of the right side parts expressions of the pattern-matching.

Below follow some examples showing pattern-matching typechecking.

```
f(x) =
  match x with
  | 0 -> false
  | 1 -> false
  | _ -> true
f : int -> bool =  <fun>
```

Typechecking the matched expression `x` doesn't impose any particular constraint. Hence `x` is assigned a type variable. Then, each pattern is typechecked and appears to be `int`. Now, each right-side part is typechecked leading to type `bool` which is the return type of the function. The last pattern is a catchall, but since it doesn't have a sum type, opening has no effect. Since this function takes `x` as parameter and this latter was inferred the type `int`, the final type of this function is as expected `int -> bool`.

```
g1(x) =
  match x with
  | { a = 1 } -> 0
  | { b = bval } -> bval
g1 : {b : int } / {a : int} -> int = <fun>
```

Typechecking the patterns, it can be seen that the matched expression must be of a type sum containing either `{ a : int }` or `{ b : 'a }`. At this point, the pattern variable `bval` is not yet constrained to any particular type from the current context, so it remains a type variable. Hence, the type of the matched expression must be, a priori, `{b : 'a} / {a : int}`. The first right side part is then typechecked leading to type `int`. The second one, thanks to the previously recorded binding of `bval` to `'a`, has type `'a` and must be compatible with `int`. Hence `'a` must be instantiated by `int`. As a result, the matched value must have type `{b : int } / {a : int}` and the result of the pattern-matching must have type `int`. Finally, we have a catchall pattern, the variable `bval`, but since its type is not a sum type, opening has no effect and types are not changed. Hence, the function `g1` is inferred of type `{b : int } / {a : int} -> int`.

```
g2(x) =
  match x with
  | { a = 1 } -> 0
  | { b = bval } -> bval
  | _ -> 42
g2 : {b : int} / {a : int} / ... -> int = <fun>
```

This example only differs from the previous one by the addition of a final catchall case. Typechecking is done in a similar way than in the previous example. The only difference is that because the catchall pattern is related to a sum type (`{b : int} / {a : int}`), opening columns has now an effect. The type of the matched value finally gets opened, under the form `{b : int} / {a : int} / ...` to reflect the fact that this pattern-matching can handle sums with a case `{ a }`, a case `{ b }` or any other case.

```
g3(x) =
  match x with
  | { a = { c } } -> { d }
  | { b = bval } -> bval
g3 : {b : {d} / ...} / {a : {c}} -> {d} / ... = <fun>
```

More subtle, the above example shows that while typechecking the patterns, the catchall pattern variable `bval` is not yet constrained, hence remains a type variable `'a`. If we have a look "in the future", we will see that right-side parts impose that `'a` is instantiated by a sum type. However, this is not yet known and the opening operation has no effect on the type currently bound to `bval`. The type of right-side parts is now inferred. The first one imposes to have a sum type containing a case `{ d }`, hence has type `{ d } / ...` (remember that a record expression is by default inferred as plugged in an opened column). The second one imposes now `'a` to be instantiated by the former type. This instantiation of `'a` propagates both in the types of the argument and of the result of the function `g3`, stating that this pattern-matching can in fact handle sums with a case `{ a : { c } }` and a case `{ b }` itself having a sum type containing "`d` or any other case". This difference then appears in the global type of the function `g3` since it shows that it returns a sum with either a case `a` or any other case being the embedded value of its argument when it is a sum of the form `{ b = ... }`.

```
g4(x) =
  match x with
  | { a = 1; b = { c = 3 } } -> 0
  | { a = 1; b = { c = 4 } } -> 0
  | _ -> 0
g4 : {a : int; b : {c : int} / ... } / ... -> int = <fun>
```

This example demonstrates the opening of sums due to catchall patterns operating in depth. In effect, the final "_" pattern is bound to a sum type `{a : int; b : {c : int} }`, i.e. a sum whose first case has a field `b` also having a sum type. Since this catchall "absorbs all the other cases" of this sum type, it absorbs others cases than `{ a ; b }` but recursively also " the other cases than `{ c }` in the type of `b`". For this reason, both sum types get opened in the argument of the function (note the apparition of the 2 column variables `/ ...`).

Pattern-matching and records

By default, when typechecking patterns, if all the patterns lead to a same record type, the inferred type is a sum type with only one case being this record type:

```
g5(x) =
  match x with
  | { a = 1 ; b = "" } -> x
```

```
  | { a = 2 ; b = "A" } -> x
  | _ -> x
g5 : {a : int; b : string} / ... -> {a : int; b : string} / ... = <fun>
v = g5({ a = 1 ; b = "" })
v : {a : int; b : string} / ... = { a = 1 ; b = "" }
_ = v.a
_do_ : int = 1
```

Since such a sum only has one case, it remains compatible with a record as shows the creation of `v` then its field access in the above sample code. However, it is possible to add constraints enforcing the matched expression to have a pure record type, i.e. a closed sum. For instance, in:

```
g6(x) =
  match x with
  | { a = 1 ; b = "" } -> x.a
  | { a = 2 ; b = "A" } -> 1
  | _ -> 2
g6 : {a : int; b : string} -> int = <fun>
```

the field access performed on `x` in the right-side part of the first case imposes `x` to be a record (see section dealing with field access), i.e. to be plugged into a closed column. This appears in the type of `g6` since is has any more column variable in spite of the catchall pattern. In effect, opening operation is performed after typechecking the patterns and before typechecking right-side parts. Hence, the field access construct enforce closing again the column that was opened due to the final catchall pattern. This is sound because if the sum was remained opened, the following example

```
g6bug(x) =
  match x with
  | { a = 1 ; b = "" } -> x.a
  | { a = 2 ; b = "A" } -> 1
  | _ -> x.a
```

would also have type `{a : int; b : string} /... -> int` which would be unsound because it would mean that in the default case, when none of the 2 first patterns matches the tested value, we return its field `x`. Passing this function a value of any form not compatible with the 2 first patterns and not containing a field `x`, for instance, `{ c = 1 }`, would fall in this default case and since no field `a` is present, the field access would fail.

Pattern-matching and elipsis

A special form of pattern, the elipsis written "..." can be used in record patterns involved with both sums and records. However, it currently has 2 different meanings depending on the context and these 2 meanings may sometimes conflict. Elipsis is used in patterns under the form:

```
match x with
  | { a = 1 ; ... } ->
```

The first meaning of the elipsis is "this pattern is an opened record". In other words, "this pattern matches records with at least all the specified fields, all other ones being of no interest". For instance:

```
h1(x) =
  match x with
  | { a = v; ... } -> v + 1
h1 : {a : int; ...} -> int = <fun>
_ = h1({ a = 3 ; b = "useless"; c = 3.14159 })
_do_ : int = 4
```

states that the function `h1` takes as argument a record having at least a field `a` of type `int` and ignores all other possible fields of the record actually passed.

The second meaning of the elipsis deals with sum cases in which only some fields of each case are useful and means "this pattern matches a case of a sum containing the specified fields that are useful and the other fields of this case are of no interest but I don't want to enumerate all of them". For instance

```
type t =
  { a : int ; b : int ; c : int ; d : int } /
  { e : int ; f : int }
type t = {a : int; b : int; c : int; d : int} / {e : int; f : int}
h2(x) =
  match x : t with
  | { a = a_val ; ... } -> a_val
  | { f = f_val ; ... } -> f_val
h2 : t -> int = <fun>
```

defines a sum type with 2 cases and several fields inside. The function `h2` performs a pattern-matching on a value of this type but is only interested in one of the fields of each case. To avoid the tedious enumeration of all the other useless cases, elipsis is used to let the typechecker determine the remaining part of the fields of each case.

However, this 2-sided aspect of the elipsis introduces tricky behavior when the context doesn't permit to guess if the elipsis is used in the context of a record or a sum type. Without enough information, the elipsis is implicitly considered as meaning an opened record type. This means that this creates, for the pattern, a record type ended by a row variable. When unifying the type of patterns and matched expression to ensure they are compatible together, if the opened record types of 2 patterns get unified, they will be along the record and not along the sum. In other words, 2 opened records will join their fields in a unique final type although 2 closed records with incompatible fields will join into a sum type with these 2 records as cases. Hence,

```
h3(x) =
  match x with
  | { a = 1; ... } -> 0
  | { b = 2; ... } -> 0
h3 : {a : int; b : int; ...} -> int = <fun>
```

having 2 opened cases, infers for them `{ a : int ; ... }` and `{ b : int ; ... }`, that get unified into {a : int; b : int; . . . }, giving this possibly surprising type to `h3`. In fact, the point is that there is not enough information in the context to guess if these 2 patterns represent 2 different "interesting" fields of a record or represent 2 cases of a sum in which we don't want to enumerate useless fields of each case. The difference with the previous example is that in the previous one, there was a type constraint on the matched expression bringing the information that the matched value had a sum type. With this extra information, the typechecker "knew" that patterns were related to a sum type and could complete the omitted fields of each case instead of considering them as 2 records to merge together.

As we stated above, the elipsis is implicitly considered as meaning an opened record type hence creates a type ended by a row variable. If when checking that patterns have compatible types unification fails unifying along the record, i.e. merging the fields of the record, then it tries to unify along the column, i.e. creating a sum case with the 2 records as cases. Consider

```
h4(x) =
  match x with
  | { a = 1; ... } -> 0
  | { b = "true" } -> 0
  | { a = _ } -> 0
h4 : {a : int} / {b : string} -> int = <fun>
```

the types inferred for each pattern is `{ a : int ; ... } / ...`, `{ b : string } / ...` and `{ a : int }`. Unifying the 2 first types can't succeed along the row because the second record type is closed, then can't acquire the field `a`. Hence unification is performed along the column, leading to a sum type `{a : int; ...} / {b : string} / ...`. Now unification is done between this type and the one of the last pattern, `{ a : int }`. These 2 types can be unified since `{a : int; ... }` and `{ a : int }` can be, instantiating and closing the ending row variable. Since there is no catchall pattern, the resulting sum is not opened, hence doesn't show any column variable.

Caution
What happens if both row and column variables appear ? The answer is that in this case, it may be possible to unify types along row or columns, i.e. to merge fields of records or to catenate records as sum cases. In some cases, there is no better choice and hence no principal type. This weakness of the type system is known, as well as the fact that unification "prefers" (first tries) to unify along the rows. That's some reasons why some type annotations are sometimes required to get out of tying errors.

Let's consider the following example:

```
h5(x) =
  match x with
  | { a = 1; ... } -> 0
  | { b = "true" } -> 0
  | _ -> 0
```

The first pattern has type `{ a : int ; ... } / ...`, the second one has type `{ b : string } / ...` and the third has type `'a`. When unifying the 2 first types to ensure they are compatible, the only solution is to unify along the columns, hence leading to a sum type: `{ a: int ; ... } / { b : string } / ...`. Unification with the last type only instantiates `'a`. Because there is a catchall pattern, all sum types remain opened, hence the global type of the patterns is finally `{ a: int ; ... } / { b : string } / ...`. We are exactly in presence of a type that can be unified either along its column or row variable, hence in presence of possibilities of non-existence of principal type. In such a case, an error is raised by the typechecker:

```
Sum type with row and column variables
{ b: string } / { a: int; 'r.a } / 'c.a
Error
Exportation failure
```

It is worthy to note that at this point, the typechecker detected an issue while working with a richer type algebra that the one available to state the signatures. In effect, this algebra temporarily tolerates sums with row and column variables but is only internally used by the typechecker. Error messages related to the internal type algebra print types a bit differently.

Caution
Because a sum type represents an enumeration of cases, possibly opened, but with each case having a fixed and finite set of fields, sums with cases containing row variables are also not legal types. For this reason, even if the richer and internal type algebra used by the typechecker temporarily tolerates such types, once typecheking ends such types, if they escape are rejected and don't fit the public type algebra in which signatures of the values are stated. Hence, the following example:

```
h6(x) =
  match x with
  | { a = 1; ... } -> 0
  | { b = "true" } -> 0
Closed sum type with row variable(s) { b: string } / { a: int; 'r.a }
Error
Exportation failure
```

is rejected since the first pattern has type `{ a : int ; ... } / ...`. The second one has type `{ b : string } / ...`. Hence, unifying these 2 types gives `{ a : int ; ... } / { b : string } / ...`. Since there is no catchall pattern, the sum doesn't remain opened hence leading to type `{ a : int ; ... } / { b : string }`. This type is a sum and it actually has a case being a row ended by a variable.

C.5.2.9 Binary conditional

Binary conditional is introduced under the form:

```
if expr1 then expr2 else expr3
```

where $expr_1$ is the tested condition, expected to be of type `bool`, $expr_2$ is the result value in case the condition holds (i.e. the evaluation of $expr_1$ yields `true`) and $expr_3$ is the result value in case the condition doesn't holds. Since this construct can return the value of either $expr_2$ or $expr_3$, both expressions must have the same type. Hence, typechecking of this construct if performed in 3 steps:

1. The condition expression $expr_1$ is typechecked and it is checked to be of type `bool`.

2. The expression $expr_2$ and $expr_3$ are typechecked, leading to types τ_2 and τ_3.
3. Types τ_2 and τ_3 are checked to be compatible, i.e. are unified in a common type τ and if so, the type of the whole expression is then τ.

Example:

```
ok = if true then 1 else 2
ok : int = 1
reok = if 1 != 2 then { A } else { B }
reok : {A : {}} / {B : {}} / ... = { A = void }
ko = if 1 then { A } else { B }
Error
Matched expression or previous patterns have type int but new pattern is
found of type { true } / 'c.a.
reko = if false then 1 else "one"
Error
Previous right-side parts of the pattern matching return type int but current
one returns type string.
Types int and string are not compatible
test(c, default) = if c then "Holds" else default
test : bool, string -> string = <fun>
```

In the first example, the condition expression `true` is correctly of type `bool`, then both `1` and `2` have type `int` hence the whole expression is well-typed and of type `int`.

In the second example, assuming that the comparison operator "different" `!=` has type `'a -> 'a -> bool`, its application to 2 integers is well-typed and of type `bool`. Hence the condition expression is well-typed. The "then" part has type `{ A } / ...`, the "else "part has type `{ B } / ...`, hence the common type obtained by unification of these 2 types is `{ A } / { B } / ...` which is the type of the whole expression.

The third example shows that because the condition expression `1` has type `int` and not `bool` the whole expression is ill-typed.

The next example shows that because both branches of the binary conditional do not have the same type (`string` versus `int`), even if the condition expression is of the correct type `bool`, the whole expression is ill-typed.

Finally, the last example exhibits the constraints imposed by the binary conditional construct, making them appearing in the type of the function using it. In effect, since the tested condition is the first parameter, `c` of the function `test`, this enforce this parameter to be of type `int`. Moreover, since the "then" branch has type `string` and since both branches are assumed to have the same type, the "else" part must also have type `string`, which requires `default`, the second parameter of `test` to be of type `string`, which is also the type of the whole conditional expression. From these constraints, we actually find that `test` has type `bool, string -> string`.

Note
Internally, the binary conditional construct is transformed in a pattern-matching on boolean values. In other words, `if c then t else e` is rewritten as `match c with { true } -> t | { false } -> e`. That's the reason why, in case of typechecking error on a binary conditional, one sometimes get a message involving pattern-matching as we could see in some above examples.

C.5.2.10 Modules

Opa provides a "light" notion of modules through a special form of records that are written between double-braces instead of simple-braces. Hence, a module expression has the form:

```
{{ [label = expr ; ...] }}
```

The main feature brought by modules compared to records is the ability to have each field separately generalized and all of them implicitly mutually recursive. Hence, a module can be seen as a record grouping several definitions instead of leaving them spanning at toplevel.

Since modules are a sort of records they are first-class values, i.e. can be passed to functions, returned as result, stored in data-structures, etc. However their type is not a simple record type. In effect, since each field, i.e. definition of the module can be generalized, the type of the definitions are especially processed before being part of the module type: if they are polymorphic then their type is turned into a type-forall (see types forall) to mimic a type scheme. The final type of a module is them a **closed** record type in which labels are bound to their respective type (possibly being type-forall), embedded in a **closed** sum (i.e. the sum has the record as unique "*case*").

Typechecking of a module is performed in 4 steps:

1. Like for record expressions, each field expression $expr_i$ is typechecked and like for a recursive identifier binding (see dedicated section), fields are considered as implicitly mutually recursive. As a result, we obtain as many types τ_i.
2. Each type τ_i is inspected, any unconstrained type variables get generalized and a type-forall τ'_i is built quantifying these variables and whose body it τ_i. The related module field is then assigned this new type τ_i. However if no such type variable exist in τ_i, then no change is done and the module field keeps its initial type τ_i.
3. A closed record type is then built, binding each $label_i$ to its related found type.
4. Finally, an closed sum type is created, plugging the above record type as its unique "*case*".

Examples:

```
M = {{
  plus(x, y) = x + y ;
  f(x) = x
}}
M : { f : forall('v0). 'v0 -> 'v0; plus : int, int -> int} / ... = { f = <fun> ; plus = < ↩
    fun> }
```

As we stated before, the type of a module is a closed record in a closed sum. The reason for having a closed record is that, like any record, a module expression enumerates all the fields this module has, and it has no more fields that those making its definition. There are 2 reasons for having a closed column. First, modules are not sum types: they do not present "different cases". Next, having an ending column variable would make it unconstrained, hence generalized. In such a case, when creatign sub-modules, the type of the sub-module would have a generalizable column variable and would naturally turn itself into a type-forall which is not the expected type for a module.

This feature, pretty related to rank 2 polymorphism, however introduces some usage restrictions. The first important point is that modules are explicitly introduced: when the syntactic construct `{{ }}` is used, a module is to be created. The second point to remind is that instantiation of type-forall is automatic: each time an expression is found having a type-forall, it is immediately specialized, replacing its generalized variables by fresh ones and removing the forall quantification (i.e. the obtained type is an instance of the type-forall's body).

Let's examine a simple example.

```
// Definition of a module type.
type Mty = {{ f : 'a -> 'a ; g : int -> int }}
type Mty = {f : forall('a). 'a -> 'a ; g : int -> int}

// Coercion to a module type
M = {{ f(x) = x ; g(x) = x + 1 }} : Mty
M : Mty = { f = <fun> ; g = <fun> }
_ = M.f(5)
_do_ : int = 5
_ = M.f("foo")
_do_ : string = "foo"
```

The first type definition introduces a module type, automatically adding the forall-types on polymorphic fields. This avoid having to explicitly state them in the type de finition. Next a module `M` is defined and its signature gets constrained by `Mty`. Note that this constraint is not mandatory but set here to show how to constrain a module and to avoid getting a verbose structural type like `{f : forall('a). 'a -> 'a; g : int -> int} = { f = <fun> ; g = <fun> ; }` we would get otherwise. Module fields access is performed like for record, via the dot-notation. The point to note is that `M.f` is a

forall-type, hence during the application to the expression `5`, `M.f` appears to have type `forall('a). 'a -> 'a`. As stated above, this forall-type get automatically instantiated into a fresh type `'b -> 'b`, and the type of the whole expression is `int` as expected. Following this point, when applied a second time, to the expression `"foo"`, this type is instantiated again into another fresh type `'c -> 'c`, hence preserving the polymorphism and allowing to apply `M.f` to a string, returning the expected `string` final type.

An important remark is that module types are strict and there is no implicit instantiation of explicit quantification. Automatic instantiation is only done when encountering a forall-type as a type, not as a subterm-of a type. Hence, coercion cannot be used to restrict the type of a module:

```
// Rejected.
_ = {{ f(x) = x ; g(x) = x + 1 }} : {{ f : int -> int ; g : int -> int }}
Error
Expression has type { f: !for all 'a . 'a -> 'a; g: int -> int } but is
coerced into { f: int -> int; g: int -> int }.
Types !for all 'a . 'a -> 'a and int -> int are not compatible
Hint:
  Error occurred through field f.

// Ok.
_ = {{ f(x:int) = x ; g(x) = x + 1 }} : {{ f : int -> int ; g : int -> int }}
_do_ : {f : int -> int; g : int -> int} = { f = <fun> ; g = <fun> }

// Rejected.
_ = M : {{ f : int -> int ; g : int -> int }}
Error
Expression has type Mty but is coerced into { f: int -> int; g: int -> int }.
Types int -> int and !for all 'a . 'a -> 'a are not compatible
Hint:
  Error occurred through field f.

// Ok.
_ = {{ f = M.f : int -> int ; g = M.g }} : {{ f : int -> int ; g : int -> int }}
_do_ : {f : int -> int; g : int -> int} = { f = <fun> ; g = <fun> }

// Ok.
_ = {{ f(x:int) = M.f(x) ; g(x:int) = M.g(x) }} : {{ f : int -> int ; g : int -> int }}
_do_ : {f : int -> int; g : int -> int} = { f = <fun> ; g = <fun> }
```

Modules can be returned as results of functions, hence providing the mean to create functors. For instance:

```
type Mty2 = {{ f : 'a -> 'a }}
type Mty2 = {f : forall('a). 'a -> 'a}
M2 = {{ f(x) = x }} : Mty2
M2 : Mty2 = { f = <fun> }

// A functor.
F(m : Mty2) = {{
  a(x) = m.f(x)
  b(x, y) = {
    f1 = m.f(1)
    f2 = m.f(x)
    f3 = m.f("string")
    f4 = m.f(y)
  }
}}
F : Mty2 ->
      {a : forall('v0). 'v0 -> 'v0;
        b : forall('v0, 'v1, rows:, cols:...).
              'v0, 'v1 -> {f1 : int; f2 : 'v0; f3 : string; f4 : 'v1} / ...} = <fun>

// Application of the functor.
```

```
M3 = F(M2)
M3 : {a : forall('v0). 'v0 -> 'v0;
       b : forall('v0, 'v1, rows:, cols:...).
             'v0, 'v1 -> {f1 : int; f2 : 'v0; f3 : string; f4 : 'v1} / ...} = { a = <fun> ; ↩
                  b = <fun> }

_ = M3.a(1)
_do_ : int = 1
_ = M3.b("toto", 18.26)
_do_ : {f1 : int; f2 : string; f3 : string; f4 : float} / ... =
  { f1 = 1 ; f2 = "toto" ; f3 = "string" ; f4 = 18.26 }
```

However, an important consequence of module types strictness is that a module cannot be passed to a function expecting a less general module or a record, for instance:

```
f(m) = {{
  a(x) = {
    f1 = m.f(x) ;
    f2 = m.f(365)
  }
}}
f : {f : int -> 'v0; ...} ->
  {a : forall(, rows:, cols:...). int -> {f1 : 'v0; f2 : 'v0} / ...} = <fun>

_ = f(M)     // Rejected because M is too general.
Error
Function was found of type
{ f: int -> 'a; 'r.a } ->
{ a: !for all 'c.a . int -> { f1: 'a; f2: 'a } / 'c.a } but application
expects it to be of type Mty -> 'b.
Types int -> 'a and !for all 'c . 'c -> 'c are not compatible
Hint:
  Error occurred through field f.

// Ok because M.f is instantiated.
_ = f({ f = M.f })
_do_ : {a : forall(, rows:, cols:...). int -> {f1 : int; f2 : int} / ...} = { a = <fun> }

// Rejected.
_ = f({{ f = M.f }})
Error
Function was found of type
{ f: int -> 'a; 'r.a } ->
{ a: !for all 'c.a . int -> { f1: 'a; f2: 'a } / 'c.a } but application
expects it to be of type { f: !for all 'b . 'b -> 'b } -> 'c.
Types int -> 'a and !for all 'b . 'b -> 'b are not compatible

// Ok.
_ = f({{ f = M.f : int -> int }})
_do_ : {a : forall(, rows:, cols:...). int -> {f1 : int; f2 : int} / ...} = { a = <fun> }
```

C.5.2.11 Field access ("dot" access)

This construct allows to access the value of a record field following the form *expr*.*field name*. As a consequence, it is expected that *expr* has a record type in which the field *field name* exists with a given type τ. The type of the whole expression is then this type τ.

```
type t('a) = { a : 'a }
type t('a) = { a : 'a }
x = { a = 3.14159 } : t(float)
```

```
x : t(float) = { a = 3.14159 }
_ = x.a
_do_ : float = 3.14159
_ = x.b
Error
Record has type t(float) but field access expected it to have type
{ b: 'a; 'r.a }.
```

As shows the last example, if the "dotted" expression doesn't have a record type containing at least the required field, the expression is ill-typed. It is important to note that the expected record type must contain at least the accessed fields and may contain any other ones as the ending row variable `'r.a` suggests.

C.5.2.12 Record update

The construct:

```
{ expr0 with field1 = expr1 ; ... ; fieldn = exprn }
```

allows creating a new record value, borrowing all the fields from $expr_0$ but replacing the values of $field_1$ to $field_n$ by those evaluated for $expr_1$ to $expr_n$. Three important points are:

- The fields updated must belong to the type of $expr_0$. This means that record update doesn't add new fields to the record. This restriction is required to keep Opa's type system consistent.
- The types of the updated fields can however change from the ones they had in $expr_0$.
- This construct only applies on records, i.e. opened or closed rows hosted by a closed column.

Let's assume that $expr_0$ has a record type with fields $field_1$ to $field_n$ and other possible fields $fields_m$ of type τ_{ms}. Expressions $expr_1$, to $expr_n$ are typechecked leading to types $\tau_1, \ldots, \tau_n$. The type of the whole expression is then the record type where $field_1$ is bound to type τ_1, ..., $field_n$ bound to type τ_n and all other fields from the type of $expr_0$ are unchanged. In particular, if the original record type was opened (resp. closed), so will be the resulting one. This is clearly shown by the type of the function `f` in the example below where both the argument and return types are opened records since the body of this function only add the constraint that the record must contain at least the field `x`.

```
v = { x = 3 ; y = 4 }
v : {x : int; y : int} / ... = { x = 3 ; y = 4 }
w = { v with x = 3.14159 }
w : {x : float; y : int} = { x = 3.14159 ; y = 4 }
f(r) = { r with x = "str" }
f : {x : 'v0; ...} -> {x : string; ...} = <fun>
x = f({ z = 5 })
Error
Function was found of type { x: 'a; 'r.a } -> { x: string; <'r.a> } but
application expects it to be of type { z: int } / 'c.a -> 'b.
Types { x: 'a; 'r.a } and { z: int } / 'c.a are not compatible
Hint:
  Field(s) x only appear(s) in the first type.
Hint:
  Field(s) z only appear(s) in the second type.
```

C.5.2.13 Type coercion

Although Opa proposes type inference, it is sometimes useful to force an expression to have a certain type instead of the one naturally guessed by the typechecker. This mostly happen when a particular signature is wanted for an expression (for instance for sake of abstraction) or when a type was structurally inferred although the user prefers to get a named type having the same structure (either for sake of shortness or because the type structurally inferred is recursive and hence must be explicitly replaced by a named type since cyclic types are not accepted in types appearing in values interfaces).

Type coercion, *expr*:*t*, typechecks *expr* leading to a type τ. Then is converts the type expression *t* leading to a second type τ'. Finally, if τ and τ' are compatible, it gives the whole expression the type τ'.

```
type t = int
type t = int
_ = 4 : t
_do_ : t = 4
_ = 4.5 : t
Error
Description: This expression has type float.
However, according to the context, it seems that it should have type t.
type ilst = { head : int ; tail : ilst } / { nil }
type ilst = {head : int; tail : ilst} / {nil}
rec f(x, y) = if x then { nil } else { head = y ; tail = f(true, y) }
Cyclic type 'a
  where 'a = { head: 'b; tail: 'a } / { nil } / 'c.a
Error
Exportation failure
rec f(x, y) : ilst = if x then { nil } else { head = y ; tail = f(true, y) }
f : bool, int -> ilst = <fun>
```

The first coercion in the above sample code show that because `t` and `int` are compatible, the value `4` being naturally of type `int` can be assigned the type `t`. The second example shows that because `4.5` has type `float` and this latter and `int` are not compatible, coercion is a failure. Finally, the last example exhibits a case where an explicit naming of a recursive type must be done. First, a type similar to a list of integer is defined, `ilst`. Then the function `f` that builds empty or one-length lists of integer is defined. Since Opa typechecking is performed in a structural way, nothing in the body of `f` indicates that it works on a particular named type. The constraints only indicates that the function returns a record type containing a field `tail` having this record type. As a result, an error is issued since cyclic types must not escape in signatures. Adding a coercion on the return type of `f`, like in the last line of code, allows the typechecker to find a named type hiding the recursion and removes the error.

Implicit type parameters in coercions

When using a parameterized type name in a coercion, it is possible to omit the arguments by which its parameters must be instantiated. The typechecker will then automatically instantiate them according to the context. Omission applies to all or none of the parameters: only omitting some of them will result in a typechecking error related to the arity of the used type name. To omit only some of the arguments, they must explicitly be replaced by some underscore ("_") characters upon matching the right arity of the type constructor.

```
type t('a, 'b) = { first : 'a ; second : 'b }
type t('a, 'b) = { first : 'a; second : 'b }
_ : t = { first = 1 ; second = "str" }
_do_ : t(int, string) = { first = 1 ; second = "str" }
_ : t(int) = { first = 1 ; second = "str" }
Error
Description: The type instanciation for t is invalid. According to the
definition of this type, it should have 2 argument(s). However, here, it is
applied to 1 argument(s).
_ : t(int, _) = { first = 1 ; second = "str" }
_do_ : t(int, string) = { first = 1 ; second = "str" }
```

In the above sample code, a record type, `t`, with 2 parameters is defined. The first coercion omits all of them. According to the context, since field `first` is assigned the value `1` and field the value `"str"` it is inferred that `'a` maps onto `int` and `'b` onto `string`. The second coercion only omits 1 of the parameters but doesn't explicitly state the omission, hence making the type `t` used with a wrong number of argument. Finally, the last example explicitly claims that the first parameter of `t` is expected to be `int` but second one is not explicitly given, hence making the `t` type constructor used with its right number of arguments and avoiding the previous error.

Type variables in coercions and generalization

Type variables appearing in coercion type expressions have precise scoping rules and may influence the generalization process, i.e. the polymorphism level of definitions. Mostly, in type annotations, type variables are identified by their name. This means

that inside a definition, several occurrences of a same type variable name implies the sharing of this type variable among these occurrences. The scope of a type variable name is:

- the current toplevel value definition,
- the current module field definition.

In other words, all along these 2 constructs, a same variable name represents a unique type variable. For instance, in:

```
f(x, y) = y : 'a
f : 'v0, 'a -> 'a = <fun>
g(x:'a, y) = y : 'a
g : 'a, 'a -> 'a = <fun>
```

the definition of `f` introduces a type variable `'a` used only once. As a result, `x` and `y` have 2 different types since they are not related. Conversely, in `g`, a new type variable `'a` is introduced (diferent from the one of `f`), but used twice, once to constrain `x` and once to constrain `y`. Since these 2 occurrences of `'a` are hosted in the same toplevel definition, they represent the same variable. As a result, the types of the 2 arguments of `g` are enforced to be the same which was not the case in `f`.

This example can be reproduced the same way in a module since the scope of a variable introduced in a module field is this module field:

```
M = {{
  f(x, y) = y : 'a
  g(x:'a, y) = y : 'a
}}
M : {
  f : forall('v0, 'a). 'v0, 'a -> 'a ;
  g : forall('a). 'a, 'a -> 'a } =
 { f = <fun> ; g = <fun> }
```

As the type-forall's suggest variables are differents (i.e. not shared) between the fields `f` and `g` since they are generalized separatly. And like in the previous example, `f` has 2 arguments with different types although `g` have its ones of the same type.

Caution
Introducing type variables in coercions **doesn't** mean that the annotated expression is polymorphic ! This only means that its type is named by this type variable. Generalization, i.e. polymorphism introduction is a separate process that arises at the end of a definition. In particular, introducing a type variable in a annotation will postpone its possible generalization upon the end of the current toplevel or module field definition. In effect, as soon as a name is introduced for a type, we can't make sure this name will be or not used later, until we get out of the definition where it is introduced. For this reason, it is not safe to generalize it as soon as we leave the definition where it appears since some later type constraints may change this variable into a non-variable type, i.e. the variable will not remain a variable hence can't be subject to generalization. Let's consider:

```
f1() =
  rec v1(x : 'a) : 'b = v1(x)
  v1
f1 :  -> ('a -> 'b) =  <fun>
f2() =
  rec v1(x : 'a) : 'b = v1(x)
  _ = v1(1)
  _ = v1(1.0)
  v1
Error
Function was found of type int -> 'a but application expects it to be of type
float -> 'b.
Types int and float are not compatible
f3() =
```

```
  rec v1(x : 'a) : 'b = v1(x)
  v2 : bool -> int = v1
  v1
f3 :  -> (bool -> int) =  <fun>
```

Let's first have a look at `f1`. It introduces 2 variables, `'a` and `'b` that finally get generalized in the final type of `f1` that really is polymorphic. However, it must be clear that these 2 type variables were **not** generalized at the end of the local definition of `v1` ! They were generalized at the end of the toplevel definition of `f1` through the generalization of the type of `f1`. In other words, inside the body of `f1`, the type of `v1` was not polymorphic because of the presence of the type variables. If we used `v1` with several types in the body of `f1`, we would get a typing error like it is the case in `f2`. The definition of `f3` also shows that `'a` and `'b` are not generalized as soon as exiting from the definition of `v1`. In effect, before returning `v1` as the result of `f3`, we define a local variable `v2` as being equal to `v1` but coercing its type. This implies a unification between the type of `v2` which is forced to be `bool -> int` and an instance of the type scheme of `v1`. But, because `v1` is not yet generalized, its type scheme is "trivial", i.e. has no generic variables. Hence, taking an instance of it implies directly sharing its scheme's body, then instantiating its non-generic variables during the unification with the type of `v2`. For this reason, `'a` becomes `bool` and `'b` becomes `int`, hence leading `f3` to finally have type `-> (bool -> int)`.

C.5.2.14 Database path

Since databases are explicitly typed at their definition, accesses to paths simply have the type they were given when the database was defined. For instance, defining a database as:

```
db /test/i : int
db /test/map : map(int, string)
```

will explicitly state that the path `/test/i` has type `int` and the path `/test/map` has type `map(int, string)`. This way, paths can only by used in contexts where their types are compatible with expected ones.

C.5.2.15 @opensums directive

Sum type definitions always lead to closed columns. In effect, such a definition enumerates all the values of the type, hence there is no reason for having a trailing column variable. In some particular cases, when an expression has a type coming from such a definition, it may be useful to alter its type to make it an anonymous opened sum in order to "plug" it in a larger sum type. Let's consider the following example:

```
type small = { a } / { b } / { c }
type small = {a} / {b} / {c}
type larger = small / { d }
type larger = small / {d}
larger_to_string(x : larger) =
  match x with
  | { a } -> "a"
  | { b } -> "b"
  | { c } -> "c"
  | { d } -> "d"
larger_to_string : larger -> string = <fun>
small_v = { a } : small
small_v : small = { a = void }
_ = larger_to_string(small_v)
Error
Function was found of type larger -> string but application expects it to be
of type small -> 'a.
Types { a } / { b } / { c } / { d } and { a } / { b } / { c } are not
compatible
Hint:
  One of the sum types may be missing the following cases of the other:
  { d }.
```

In the above example, the type `larger` clearly contains all the values of the type `small`, and even more. The function `larger_to_string` operates on all these values, hence is also able to operate on the type `small`. However, applying it to a value of type `small` fails. The reason is that the type `small` is a closed sum. Hence it can't unify with `larger` since it has no column variable to acquire the field `c`. To workaround this issue, the `@opensums` directive can be applied on the expression `small_v` in order to replace its type `small` by the structure of this type in which all columns are turned opened (i.e. ending by a column variable).

```
_ = larger_to_string(@opensums(small_v))
_do_ : string = "a"
```

This directive only has an effect when applied to an expression having a sum type. Applying it to another type will not change the type unless it is a type variable. In effect, type constraints are discovered in an arbitrary order by the typechecker. At some point, it may arise that the type of an expression is not yet complete, hence still remain a type variable even if later it will appear to be a sum type. If the directive is applied to such an expression, it can't yet have any effect although if the type constraints were acquired sooner it would have an effect. When later the type variable may get instantiated by a sum type, it will be too late to open it since the directive expression will have been already processed. To prevent such non-commutative behavior, an error message is issued requesting to add a type constraint on the expression in order to be able to have its type before applying the directive.

```
f(x) = @opensums(x)
Error
Expression argument of the @opensums directive has a type that can't be subtype of another
type (sum or record).
Hint:
  Its type is currently unconstrained. Consider explicitly adding a type constraint toward
a sum type to precise the expression's type.
```

It is important to note that this directive recursively (i.e. searching in depth) opens all the sums present in the type of its argument, except on function types where it only descends on the result type and not on the arguments types. In particular, named types are examined and their structure is also recursively traversed by the opening operation. It is worthy to make clear that because opening a sum type changes its nature, if an expression appears to have a named type being a sum, once opened, it won't have anymore this named type, but instead, the structure of this named type in which sums are now opened.

C.5.2.16 @openrecord directive

As presented in the section dedicated to the record update construct, fields borrowed from a record must exist in this record: it is forbidden to add a new field. This restriction is imposed for sake of consistency of Opa's type system. However, when 2 different record types have numerous fields in common, writing a function transforming a value of the first type to a value of the second one, we would like not to have to "*manually*" map the common fields' values 2 by 2 between the 2 records. For instance:

```
type house = { number : int ; street : string ; city : string ; zip : int }
type rented_house = { number : int ; street : string ; city : string ;
  zip : int ; start_date : int ; end_date : int }
```

defines 2 types, the first one describing houses in general and the second one describing houses being rented. The only difference is that the latter has 2 more fields than the first one. When modelling the fact that a house gets rented, we would like to write a function like:

```
rent(h : house, start_day, stop_day) : rented_house =
 { h with start_date = start_day ; end_date = stop_day }
```

However, above restriction forbids the addition of new fields as reported by the error message:

```
Error
Record to update has type house but extension requires it to have type
{ end_date: 'a; 'r.a }.
Types { end_date: 'a; 'r.a } and
{ city: string; number: int; street: string; zip: int }[red] are not compatible
Hint:
```

```
  Field(s) end_date only appear(s) in the first type.
Hint:
  Field(s) city number street zip only appear(s) in the second type.
```

One solution would then be to write this function under the form:

```
rent(h : house, start_day, stop_day) : rented_house =
  { number = h.number ; street = h.street ; city = h.city ; zip = h.zip ;
    start_date = start_day ; end_date = stop_day }
```

which is very tedious when the number of common fields is important. As an alternative it is possible to explicitly request a record type to get opened, hence adding it a fresh row variable to accept addition of new fields by the directive `@openrecord`. The renting function can then be written as:

```
rent(h : house, start_day, stop_day) : rented_house =
 { @openrecord(h) with start_date = start_day ; end_date = stop_day }
```

and finally gets the expected type: `house, int, int -> rented_house`. The `@openrecord` directive can only be applied on an expression having a record type. Applying it to another type will raise an error depending on the encountered type.

First, when the type is a variable, for the same kind of reason than explained about the @opensums directive, it is requested to add a type constraint to explicitly precise the type as shown in the following example:

```
f(x) = @openrecord(x)
Error
Expression argument of the @openrecord directive has a type that can't be subtype of
another type (record).
Hint:
  Its type is currently unconstrained. Consider explicitly adding a type constraint
toward a record type to precise the expression's type.
```

Second, for any other type not being a record or a variable the default error message is issued without any hint:

```
_ = @openrecord(1.0)
Error
Expression argument of the @openrecord directive has a type that can't be subtype of a  ↩
    record type.
```

It is important to note that, conversely to the `@opensums` directive, `@openrecord` doesn't recursively traverse types: it only acts on the toplevel structure of the type being expected, as previously explained, a record type.

C.6 Dealing with recursive types

By default, Opa typechecker infers types in a structural way. This means that, without user coercions, it determines the types by guessing their structure rather by giving them names. This is pretty satisfactory since several named types can have a same structure and there is no reason the typechecker decides itself to give a name a priori to a type structure. For instance, consider:

```
type t = { a } / { b } / { c }
type t = { a } / { b } / { c }
type u = { a } / { z } / { c }
type u = { a } / { z } / { c }
f(x) =
  match x with
  | { a } -> "a"
  | { c } -> "c"
  | _ -> "something else"
f : {c} / {a} / ... -> string = <fun>
```

the function `f` has no reason to be considered as taking an argument of type `t` more than of type `u`. The typechecker inferred a structural type, i.e. the shape of the type. If we want `f` to take explicitly a particular named type, it must be coerced by `f(x : t)` or `f(x : u)`. The fact that types are inferred structurally is powerful since `f` can naturally be used with a value of type `t` or `u` since both of them have a compatible structure:

```
_ = f({ a } : t)
_do_ : string = "a"
_ = f({ c } : u)
_do_ : string = "c"
```

However, this also has drawbacks. During type inference, the typechecker knows how to deal with, i.e. build if required, cyclic (recursive) types. However such types can't escape in values signatures. They must get mapped onto a named type, whose definition is obviously recursive, who hides the structural recursion. This may require the user to add some type annotations, i.e coercions, to prevent these recursive types from remaining in the final signature of the definition which leads to an error message.

```
rec len(l) =
  match l with
  | { end } -> 0
  | { head = _ ; tail = t } -> 1 + len(t)
Cyclic type 'a
  where 'a = { head: 'b; tail: 'a } / { end }
Error
Exportation failure
```

In the above example, the system determined that `len` is a function taking a sum type with 2 cases, the second one containing a field, `tail` being of this sum type. Hence, the type is really cyclic as reported in the error message. To avoid such a recursive type, one must define a named type having an equivalent structure:

```
type mylst('a) = { end } / { head : 'a ; tail : mylst('a) }
```

allowing to "name" the recursion. From now, adding a type annotation in the source of `len` will give the typechecker a clue "which type is used here" and will allow replacing the cyclic parts of the structure by the type name.

```
rec len(l : mylst('a)) =
  match l with
  | { end } -> 0
  | { head = _ ; tail = t } -> 1 + len(t)
len : mylst('a) -> int = <fun>
```

It could have been possible to put the type constraint at some other points, like:

```
rec len(l) =
  match l with
  | { end } : mylst('a) -> 0
  | { head = _ ; tail = t } -> 1 + len(t)

rec len(l) =
  match l with
  | { end } -> 0
  | { head = _ ; tail = t : mylst('a) } -> 1 + len(t)

rec len(l) =
  match l with
  | { end } -> 0
  | { head = _ ; tail = t } -> 1 + len(t : mylst('a))
```

but a good practice is to put constraints directly in the function's header, the closest to its parameters and return types definition. Hence, the fully annotated definition of `len` according to this good practice would be:

```
rec len(l : mylst('a)) : int =
  match l with
  | { end } -> 0
  | { head = _ ; tail = t } -> 1 + len(t)
```

where the return type is stated after the parameters of the function and before its body.