Opa Manual

by the Opa Team

Opa Manual

WIP material

May 30, 2012

Contents

A tour of Opa

Opa is a new generation of open source web development platform that lets you write *secure* and *scalable* web applications using a single technology. Throughout the pages of this book, we will introduce you to the many features of Opa. It will help if you have some knowledge of programming (any language should do) and web technology (HTML and CSS in particular).

A single language

Opa is a single programming language for writing web applications. In particular, client and server code (usually called frontend and backend) are both written in Opa.

You can write a complete program without thinking about the client-server distinction and the Opa compiler will distribute the code as needed for you and take care of all the communication. Should you need to tweak the choices made by the compiler (for instance to improve the application performance) it's very easy with simple keywords like `client`, `server` and more for fine-tuning.

```
 // Opa decides
function client_or_server(x, y) { ... }
 // Client-side
client function client_function(x, y) { ... }
 // Server-side
server function server_function(x, y) { ... }
```

The database code can also be written directly in Opa. Opa supports the major NoSQL databases MongoDB and CouchDB, and has its own internal database as well. The latter, requires no configuration at all and is recommended, especially for beginners.

Easy workflow

To write an application, first type the code in your favorite editor. The simplest "Hello, world" application in Opa is written in just a few lines:

```
Server.start(
   Server.http,
   { page: function() { <h1>Hello, world</h1> }
   , title: "Hello, world"
   }
)
```

The program can be compiled and run with the following single command line: `opa hello.opa --`[sh] // –parser js-like

The resulting application can be opened in your favorite browser at http://localhost:8080

Familiar syntax

Opa new default syntax is inspired by popular programming languages: C, JavaScript and others. Below is an extract of a real Opa program:

```
function createUser(username, password) {
    match (findUser(username)) {
    case {none}:
        user =
            { username: username
              , fullname: ""
              , password: Crypto.Hash.sha2(password)
            };
        saveUser(user);
    default:
        displayMessage("This username exists");
    };
    Client.goto("/login");
}
```

Opa however extends the classical syntax with advanced features specific to the web. HTML fragments can be inserted directly without quotes: `line = <div id="foo">bar</div>;`[opa]

CSS selectors readily available: `selector = #foo;`[opa]

And a pointer-like syntax allows to apply a given content to a selector: `*selector = line;`[opa]

Opa provides event-driven programming. For instance, running a function when an event is triggered is accomplished in the following way: `function action(_) { #foo = <div id="bar" />; } ... <div onclick={action} />`

The best place to start looking at the features of Opa and their syntax is the reference card.

Static typing

One of the most important features of Opa is its typing system. Although Opa may look like and has many advantages of dynamic programming languages, it is a compiled language which relies on a state-of-the-art type system.

Opa checks the types at compile time, which means that no type error can happen at runtime. For instance, the following code `foo = 1 + "bar";`[opa] raises the following error at compile time: `"Types int and string are not compatible"`[sh]

Unlike C or Java, you don't have to annotate types yourself as Opa features almost complete type inference. For instance, you can just write:

```
function foo(s) {
   String.length(s);
}
function bar(x, y) {
   foo(x) + y;
}
```

and the Opa compilers automatically infers the types, as if you've written:

```
int function foo(string s) {
   String.length(s);
}
int function bar(string x, int y) {
   foo(x) + y;
}
```

This system will become your wingman while you code. For instance, we will present four types of errors that are caught at compile time. The examples are taken from a real work on Opa program named webshell, available at http://github.com/hbbio/webshell.

If you write:

`element = <div> <span>{prompt({none})}</span> <span>{expr} </div> <div>{Calc.compute(expr)}</div>;` The compiler will tell you that there is an _"open and close tag mismatch vs "_.

If you write:

`case {some: 13}: #status = "Enter"; callback(get()); case {some: 37}: #status = "Left"; move({lef}); case {some: 38}: #status = "Up"; move({up}); case {some: 39}: #status = "Right"; move({right});` The compiler will tell you that the type of this function is not right. *You are using a type { `lef` } when a type { `left` } `or` { `right` } `or` { `rightmost` } `or` { `up` } `or` { `down` } is expected.* The latter type is not declared anywhere in the code, but rather was inferred by the Opa compiler from the rest of the code.

If you write:

`previous = Dom.get_content(#precaret); #precaret = String.sub(0, String.lenght(previous) - 1, previous); #postcaret += String.get(String.length(previous) - 1, previous);` the compiler will tell you that `String` module has no `lenght` function and will suggest that maybe you meant `int function length(string)` instead?

If you write:

`previous = Dom.get_content(#postcaret); #postcaret = String.sub(1, String.length(previous) - 1, previous); #precaret =+ String.get(previous);` the compiler will tell you that `String.get` is a `string function(int, string)` but application uses it as `function(string)`, meaning that you forgot the first argument which is an integer.

Opa type system not only manages basic types but complex data-structures, functions and even modules! Check the following chapter for a full presentation.

Database

Opa has support for MongoDB and CouchDB, as well as its own database engine. The latter requires no configuration and is recommended for starting new programs.

Database values are declared by stating their type: `database type /path;`[opa] for instance `database int /counter;`[opa]

In the line above, `/counter` is called a *path*, as accessing stored values bears similarities to browsing a filesystem. Getting a value from the database is simply accomplished with: `/counter`[opa] while storing (or replacing) a value with: `/path <- value`[opa]

You can store complex datastructures in the database, like maps. A map is a datastructure that associates a value to each key. The path system recognize such datastructures and allows to specify a key directly in the path. For instance, you can write: `database stringmap(string) /dictionary; ... /dictionary[key]; ... /dictionary[key] <- value; ...`

Summary

Can you guess what the following code does?

[opa|fork=hello-opa|run=http://hello-opa.tutorials.opalang.org]file://hello-opa/hello-opa.opa

Try `Run` above to find out.

Going further

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

If you have any question or feedback, do not hesitate to contact us. A few ways to get in touch:

- Opa forum;
- 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);
- Follow Opa on Twitter (@opalang) or Facebook or Google+ if you want to be always up to date.

We will be there. Let's together transform the way web development is done!

A tour of Opa

Opa is a new generation of open source web development platform that lets you write *secure* and *scalable* web applications using a single technology. Throughout the pages of this book, we will introduce you to the many features of Opa. It will help if you have some knowledge of programming (any language should do) and web technology (HTML and CSS in particular).

A single language

Opa is a single programming language for writing web applications. In particular, client and server code (usually called frontend and backend) are both written in Opa.

You can write a complete program without thinking about the client-server distinction and the Opa compiler will distribute the code as needed for you and take care of all the communication. Should you need to tweak the choices made by the compiler (for instance to improve the application performance) it's very easy with simple keywords like `client`, `server` and more for fine-tuning.

```
 // Opa decides
function client_or_server(x, y) { ... }
 // Client-side
client function client_function(x, y) { ... }
 // Server-side
server function server_function(x, y) { ... }
```

The database code can also be written directly in Opa. Opa supports the major NoSQL databases MongoDB and CouchDB, and has its own internal database as well. The latter, requires no configuration at all and is recommended, especially for beginners.

Easy workflow

To write an application, first type the code in your favorite editor. The simplest "Hello, world" application in Opa is written in just a few lines:

```
Server.start(
   Server.http,
   { page: function() { <h1>Hello, world</h1> }
   , title: "Hello, world"
   }
)
```

The program can be compiled and run with the following single command line:
`opa hello.opa --`[sh] // –parser js-like

The resulting application can be opened in your favorite browser at http://localhost:8080

Familiar syntax

Opa new default syntax is inspired by popular programming languages: C, JavaScript and others. Below is an extract of a real Opa program:

```
function createUser(username, password) {
    match (findUser(username)) {
```

```
    case {none}:
        user =
            { username: username
              , fullname: ""
              , password: Crypto.Hash.sha2(password)
            };
        saveUser(user);
    default:
        displayMessage("This username exists");
    };
    Client.goto("/login");
}
```

Opa however extends the classical syntax with advanced features specific to the web. HTML fragments can be inserted directly without quotes: `line = <div id="foo">bar</div>;`[opa]

CSS selectors readily available: `selector = #foo;`[opa]

And a pointer-like syntax allows to apply a given content to a selector: `*selector = line;`[opa]

Opa provides event-driven programming. For instance, running a function when an event is triggered is accomplished in the following way: `function action(_) { #foo = <div id="bar" />; } ... <div onclick={action} />`

The best place to start looking at the features of Opa and their syntax is the reference card.

Static typing

One of the most important features of Opa is its typing system. Although Opa may look like and has many advantages of dynamic programming languages, it is a compiled language which relies on a state-of-the-art type system.

Opa checks the types at compile time, which means that no type error can happen at runtime. For instance, the following code `foo = 1 + "bar";`[opa] raises the following error at compile time: `"Types int and string are not compatible"`[sh]

Unlike C or Java, you don't have to annotate types yourself as Opa features almost complete type inference. For instance, you can just write:

```
function foo(s) {
   String.length(s);
}
function bar(x, y) {
   foo(x) + y;
}
```

and the Opa compilers automatically infers the types, as if you've written:

```
int function foo(string s) {
   String.length(s);
}
int function bar(string x, int y) {
   foo(x) + y;
}
```

This system will become your wingman while you code. For instance, we will present four types of errors that are caught at compile time. The examples are taken from a real work on Opa program named webshell, available at http://github.com/hbbio/webshell.

If you write:

`element = <div> <span>{prompt({none})}</span> <span>{expr} </div> <div>{Calc.compute(expr)}</div>;` The compiler will tell you that there is an _"open and close tag mismatch vs "_.

If you write:

`case {some: 13}: #status = "Enter"; callback(get()); case {some: 37}: #status = "Left"; move({lef}); case {some: 38}: #status = "Up"; move({up}); case {some: 39}: #status = "Right"; move({right});` The compiler will tell you that the type of this function is not right. *You are using a type* { `lef` } *when a type* { `left` } `or` { `right` } `or` { `rightmost` } `or` { `up` } `or` { `down` } *is expected.* The latter type is not declared anywhere in the code, but rather was inferred by the Opa compiler from the rest of the code.

If you write:

`previous = Dom.get_content(#precaret); #precaret = String.sub(0, String.lenght(previous) - 1, previous); #postcaret += String.get(String.length(previous) - 1, previous);` the compiler will tell you that `String` module has no `lenght` function and will suggest that maybe you meant `int function length(string)` instead?

If you write:

`previous = Dom.get_content(#postcaret); #postcaret = String.sub(1, String.length(previous) - 1, previous); #precaret =+ String.get(previous);` the compiler will tell you that `String.get` is a `string function(int, string)` but application uses it as `function(string)`, meaning that you forgot the first argument which is an integer.

Opa type system not only manages basic types but complex data-structures, functions and even modules! Check the following chapter for a full presentation.

Database

Opa has support for MongoDB and CouchDB, as well as its own database engine. The latter requires no configuration and is recommended for starting new programs.

Database values are declared by stating their type: `database type /path;`[opa] for instance `database int /counter;`[opa]

In the line above, `/counter` is called a *path*, as accessing stored values bears similarities to browsing a filesystem. Getting a value from the database is simply accomplished with: `/counter`[opa] while storing (or replacing) a value with: `/path <- value`[opa]

You can store complex datastructures in the database, like maps. A map is a datastructure that associates a value to each key. The path system recognize such datastructures and allows to specify a key directly in the path. For instance, you can write: `database stringmap(string) /dictionary; ... /dictionary[key]; ... /dictionary[key] <- value; ...`

Summary

Can you guess what the following code does?

[opa|fork=hello-opa|run=http://hello-opa.tutorials.opalang.org]file://hello-opa/hello-opa.opa

Try `Run` above to find out.

Going further

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

If you have any question or feedback, do not hesitate to contact us. A few ways to get in touch:

- Opa forum;
- 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);
- Follow Opa on Twitter (@opalang) or Facebook or Google+ if you want to be always up to date.

We will be there. Let's together transform the way web development is done!

//[[Getting_Opa]] 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, binary packages of Opa are available for Mac OS X and Linux (Ubuntu/Debian-based, 64-bit architectures only). We do not provide binary packages for other Unixes, FreeBSD, and 32-bit variants, but you should be able to build Opa from sources on these systems (see below). A Windows port is in the works and Opa should also work on a number of other operating systems but these platforms are not supported for the moment.

Installing Opa

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

Mac OS X

- If you have not done so yet, install the Xcode tools, provided by Apple with your copy of Mac OS X. If you do not have Xcode, you can either download Xcode 4 or Xcode 3 (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.

{block}[WARNING] Starting with Xcode 4.3, Apple does not install command line tools by default anymore, so after Xcode installation, go to Preferences > Downloads > Components > Command Line Tools and click Install. You can also directly download Command Line Tools for Xcode without getting Xcode. {block}

- Download Opa for Mac OS 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`[sh]

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.

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`[sh]

and the documentation and examples will be in

`/usr/share/doc/opa`[sh]

Arch Linux

A community package is a work 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.

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]`[sh]

* *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`

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 Mac OS X, you will need:

- GNU coreutils, wget, md5sha1sum and gsed (or gnu-sed) available via MacPorts or Homebrew
- The Xcode suite, provided by Apple with your copy of Mac OS X (prior to Mac OS X 10.7 Lion).
- If you do not have Xcode, you can either download Xcode 4 or Xcode 3 (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.

{block}[WARNING] Starting with Xcode 4.3, Apple does not install command line tools by default anymore, so after Xcode installation, go to Preferences > Downloads > Components > Command Line Tools and click Install. You can also directly download Command Line Tools for Xcode without getting Xcode. {block}

Once these dependencies are satisfied, take the following steps:

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

`git clone https://github.com/MLstate/opalang.git`[sh]

- In the same terminal, enter

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

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

Do not forget to change your PATH variable if needed after that.

{block}[TIP] We also provide a script that will help you install all dependencies needed for building Opa. In this case, the only dependencies should be (tested on clean Ubuntu 11.10) : - git (to download the source) - m4 - libssl 0.9.8 or later (dev version) - zlib 1.2.3.4 or later (dev version)

In your opalang diectory, run: `dependencies/installation_helper.sh --prefix SOME_DIRECTORY`.

You will need to update your PATH variable after that.

{block}

Setting up your editor

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

Emacs

On Mac OS 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-classic-mode "/Library/Application Support/Emacs/site-lisp/opa-mode/opa-mode.
(autoload 'opa-js-mode "/Library/Application Support/Emacs/site-lisp/opa-mode/opa-js-mode.el
(add-to-list 'auto-mode-alist '("\\.opa$" . opa-js-mode))
(add-to-list 'auto-mode-alist '("\\.js\\.opa$" . opa-js-mode))
(add-to-list 'auto-mode-alist '("\\.classic\\.opa$" . opa-classic-mode))
```

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

```
(autoload 'opa-js-mode "/usr/share/opa/emacs/opa-js-mode.el" "OPA JS editing mode." t)
(autoload 'opa-classic-mode "/usr/share/opa/emacs/opa-mode.el" "OPA CLASSIC editing mode." t
(add-to-list 'auto-mode-alist '("\\.opa$" . opa-js-mode)) ;; <-- Set the default mode here
(add-to-list 'auto-mode-alist '("\\.js\\.opa$" . opa-js-mode))
(add-to-list 'auto-mode-alist '("\\.classic\\.opa$" . opa-classic-mode))
```

This allows for both Opa syntaxes, Javascript-like and the Classic mode. You can have both in the same editor but not in the same buffer, use the line shown here to set the default mode.

{block}[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:

"'{.lisp} (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) "' {block}

Vim

If you are running Linux (resp. Mac OS 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.

{block}[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. {block}

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.

Other editors

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

//[[Getting_Opa]] 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, binary packages of Opa are available for Mac OS X and Linux (Ubuntu/Debian-based, 64-bit architectures only). We do not provide binary packages for other Unixes, FreeBSD, and 32-bit variants, but you should be able to build Opa from sources on these systems (see below). A Windows port is in the works and Opa should also work on a number of other operating systems but these platforms are not supported for the moment.

Installing Opa

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

Mac OS X

- If you have not done so yet, install the Xcode tools, provided by Apple with your copy of Mac OS X. If you do not have Xcode, you can either download Xcode 4 or Xcode 3 (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.

{block}[WARNING] Starting with Xcode 4.3, Apple does not install command line tools by default anymore, so after Xcode installation, go to Preferences > Downloads > Components > Command Line Tools and click Install. You can also directly download Command Line Tools for Xcode without getting Xcode. {block}

- Download Opa for Mac OS 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`[sh]

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.

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`[sh]

and the documentation and examples will be in

`/usr/share/doc/opa`[sh]

Arch Linux

A community package is a work 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.

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]`[sh]

* *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`

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 Mac OS X, you will need:

- GNU coreutils, wget, md5sha1sum and gsed (or gnu-sed) available via MacPorts or Homebrew
- The Xcode suite, provided by Apple with your copy of Mac OS X (prior to Mac OS X 10.7 Lion).
- If you do not have Xcode, you can either download Xcode 4 or Xcode 3 (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.

{block}[WARNING] Starting with Xcode 4.3, Apple does not install command line tools by default anymore, so after Xcode installation, go to Preferences > Downloads > Components > Command Line Tools and click Install. You can also directly download Command Line Tools for Xcode without getting Xcode. {block}

Once these dependencies are satisfied, take the following steps:

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

`git clone https://github.com/MLstate/opalang.git`[sh]

- In the same terminal, enter

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

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

Do not forget to change your PATH variable if needed after that.

{block}[TIP] We also provide a script that will help you install all dependencies needed for building Opa. In this case, the only dependencies should be (tested on clean Ubuntu 11.10) : - git (to download the source) - m4 - libssl 0.9.8 or later (dev version) - zlib 1.2.3.4 or later (dev version)

In your opalang diectory, run: `dependencies/installation_helper.sh --prefix SOME_DIRECTORY`.

You will need to update your PATH variable after that.

{block}

Setting up your editor

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

Emacs

On Mac OS 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-classic-mode "/Library/Application Support/Emacs/site-lisp/opa-mode/opa-mode.
(autoload 'opa-js-mode "/Library/Application Support/Emacs/site-lisp/opa-mode/opa-js-mode.el
(add-to-list 'auto-mode-alist '("\\.opa$" . opa-js-mode))
(add-to-list 'auto-mode-alist '("\\.js\\.opa$" . opa-js-mode))
(add-to-list 'auto-mode-alist '("\\.classic\\.opa$" . opa-classic-mode))
```

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

```
(autoload 'opa-js-mode "/usr/share/opa/emacs/opa-js-mode.el" "OPA JS editing mode." t)
(autoload 'opa-classic-mode "/usr/share/opa/emacs/opa-mode.el" "OPA CLASSIC editing mode." t
(add-to-list 'auto-mode-alist '("\\.opa$" . opa-js-mode)) ;; <-- Set the default mode here
(add-to-list 'auto-mode-alist '("\\.js\\.opa$" . opa-js-mode))
(add-to-list 'auto-mode-alist '("\\.classic\\.opa$" . opa-classic-mode))
```

This allows for both Opa syntaxes, Javascript-like and the Classic mode. You can have both in the same editor but not in the same buffer, use the line shown here to set the default mode.

{block}[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:

“‘{.lisp} (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) “‘ {block}

Vim

If you are running Linux (resp. Mac OS 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.

{block}[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. {block}

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.

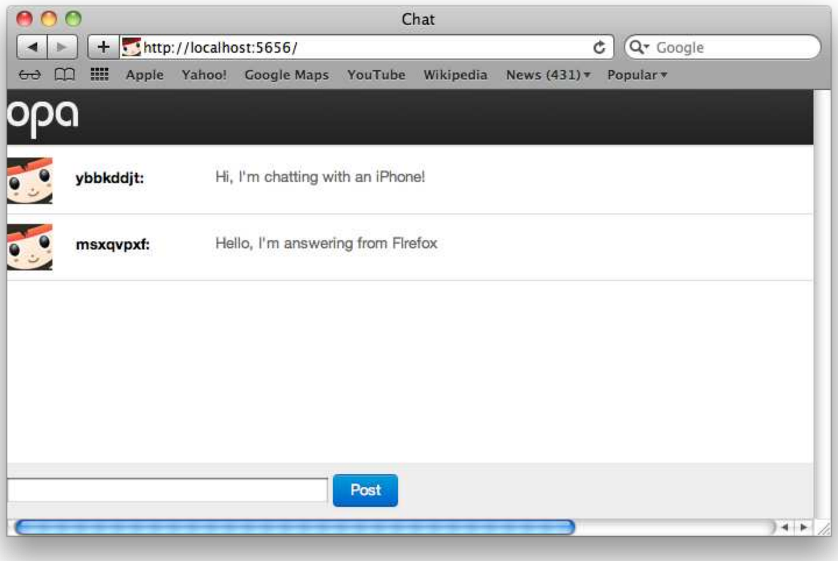
Other editors

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

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.


Chat
http://localhost:5656/
Google
Apple Yahoo! Google Maps YouTube Wikipedia News (431) Popular
opa
Post

ybbkddjt: Hi, I'm chatting with an iPhone!

msxqvpxf: Hello, I'm answering from Firefox

Figure 1: Final version of the Hello chat application

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.

Setting up communication

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 = `{`string author, string text`}[opa]

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

{block}[TIP] ### 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. {block}

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`[sh]

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. 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:

{block}[TIP] ### 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. {block}

`Network.network(message) room = Network.cloud("room")`[opa]

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.

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:

Skeleton of the user interface (incomplete)

```
<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.

{block}[TIP] ### 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 see later), apply to it functions accepting type `xhtml`, or use it as the body of a function. {block}

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

Skeleton of the user interface factorized as a function (still incomplete)

```
function 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` .

{block}[TIP] ### 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`.

The main syntax for defining a function is as follows: `function f(x1, x2, x3)` { `fun_body` }

Similarly, for a function with no argument, you can write

`function f()` { `fun_body` } 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()`. {block}

{block}[WARNING] ### Function syntax // FIXME: check In `function f(x1, x2, x3)` { `fun_body` } there is no space between `f` and `(`. Adding a space changes the meaning of the extract and would cause an error during compilation. {block}

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

Skeleton of the user interface with an arbitrary name (still incomplete)

```
function 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.

Sending and receiving

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

- at start-up, the application should *join* the room;
- whenever a message is broadcasted to the room, we should display it;

- 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.

Broadcasting a message to the room

```
function broadcast(author) {
    text = Dom.get_value(#entry);
    message = ~{author, text};
    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.

{block}[TIP] ### 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 variant of the selection operator is `#{id}`, which selects the DOM element whose id is the value of `id` (so `id` must be of type `string`). {block}

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: it can be abbreviated with {`~author, ~text`} or if all fields are constructed with such abbreviation even with: `~`{`author, text`}. So we could have written just as well:

Broadcasting a message to the room (variant)

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

{block}[TIP] ### 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. {block}

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:

Updating the user interface when a message is received

```
function user_update(message x) {
    line = <div>{x.author}: {x.text}</div>;
    #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 uses the special `#ID =+ HTML` construction to append the contents of `line` at the end of box with the id `conversation` that we have defined earlier. Instead of using the `=+` operator one can also use `+=` to *prepend* the element at the beginning of a given DOM node or simply `=` to *replace* the node with a given content.

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`.

{block}[TIP] ### 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. {block}

We are now ready to connect interactions.

Connecting interactions

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

Skeleton of the user interface connected to `broadcast` (still incomplete)

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

{block}[TIP] ### 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 online Opa API documentation by searching for entry `Dom.event`. {block}

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

Skeleton of the user interface now connected to everything (final version)

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

{block}[TIP] ### 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. {block}

Bundling, building, launching

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

{block}[TIP] ### 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. {block}

The server (first version)

```
Server.start(Server.http, {title: "Chat", page: start})
```

This extract launches a new HTTP server. For that the special `Server.start` construction is used.

{block}[TIP] ### About `Server.start` `Server.start` is, in a way, an equivalent of a `main` function in C/Java/..., as it's the entry point of the program. However, in Opa instead of just being a function to be executed when a program is started, `Server.start` starts an HTTP server to serve resources to the clients.

We will present some ways of defining servers below. You can also look up the `Server` module in online Opa API and learn more there. {block}

This is the type of the `Server.start` function:

`void Server.start(Server.conf configuration, Server.handler handler)`[opa]

It takes two parameters: configuration and a handler which essentialy defines a service.

Configuration, `Server.conf`, is just a simple record, which defines the port on which the server will run, it's netmask, used encryption and name. Most often you will use `Server.http` or `Server.https`; possibly customizing those values depending on your needs.

In this case the server is constructed using {`title: ..., page: ...`} record which builds a one-page server given a function `page` to generate this page and a `title`.

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

We have seen compilation already:

Compiling Hello, Chat

`opa hello_chat.opa`[sh]

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`[sh]

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

{block}[TIP] ### 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. {block}

{block}[TIP] ### 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. {block}

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

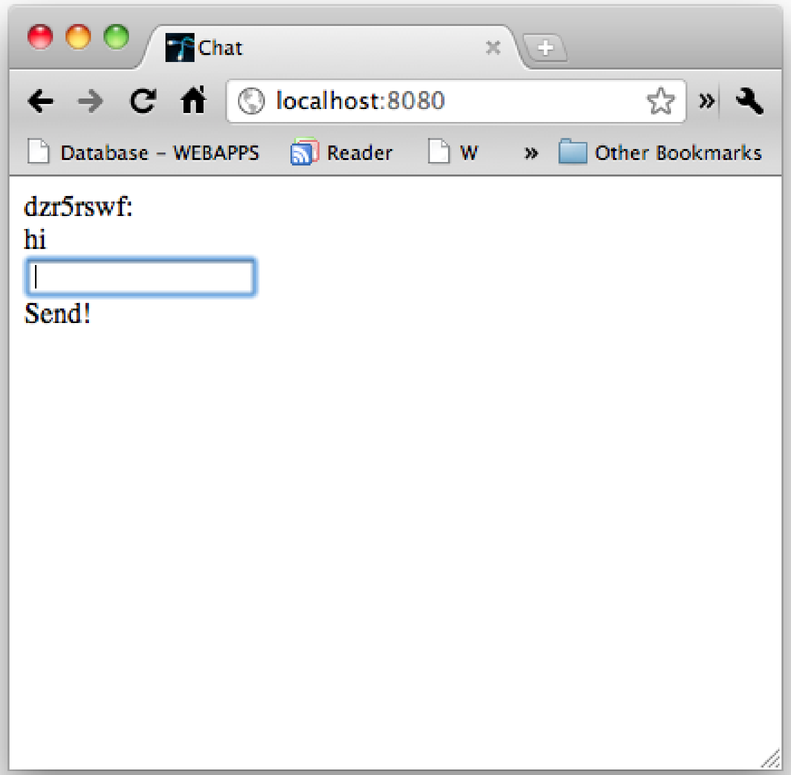


Figure 2: Resulting application, missing style

Perhaps it is time to add some style.

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] (https://developer.mozilla.org/En/CSS) 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`[opa]

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):

Main user interface

```
<div class="topbar"><div class="fill"><div class="container"><div id=#logo /></div></div></d
<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:

Function to update the user interface when a message is received

```
function user_update(message x) {
    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>;
    #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:

Contents of file `resources/chat.css`

[css]file://hello_chat/resources/chat.css

Create a directory called `resources` and save this file as `resources/chat.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 extend our `Server.start` construct by instructing it to access the directory and to use our stylesheet. We can achieve by supplying `Server.start` with a record defining our server. More forms of records are allowed, see the definition of `Server.start` in the online API.

The server (final version)

```
Server.start(
    Server.http,
    [ { resources: @static_resource_directory("resources") }
      , { register: { css: ["resources/chat.css"] } }
      , { title: "Chat", page: start }
    ]
)
```

In this extract, we have instructed the Opa *compiler* to: * embed the files of directory `resources` and offer them to the browser * use a custom resource (CSS) located at `resources/chat.css` and * start a one-page application with title *Chat* and the content of the page generated by the +start+ function.

{block}[TIP] ### 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`. {block}

{block}[TIP] ### 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 `@`. {block}

While we are adding directives, let us take this opportunity to inform the compiler that the chatroom definition `room` should be visible and directly accessible for the clients.

`exposed @async room = Network.network(message)`[opa]

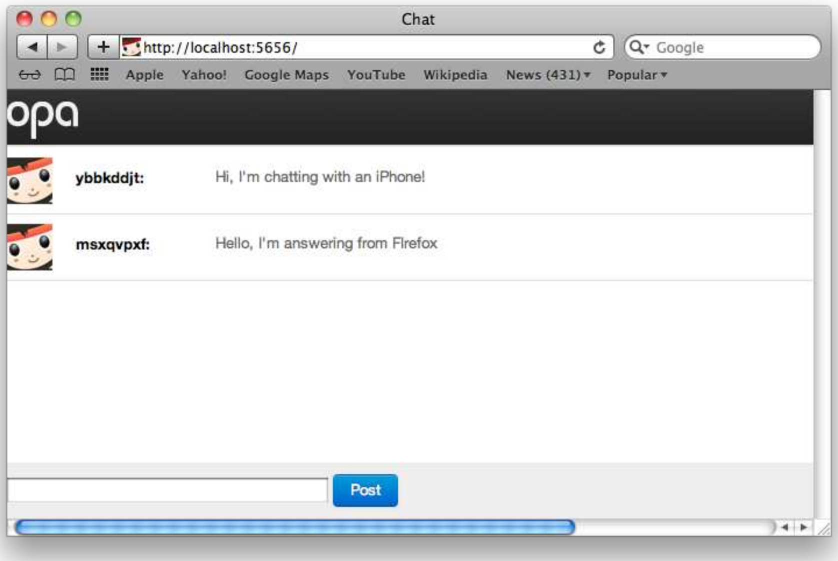
This will considerably improve the speed of the chat.

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:

The complete application

[opa|fork=hello_chat|run=http://chat.opalang.org]file://hello_chat/hello_chat.opa



ybbkddjt: Hi, I'm chatting with an iPhone!

msxqvpxf: Hello, I'm answering from Firefox

Figure 3: Final version of the Hello chat application

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.

Questions

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.

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.

Where is my `return`?

You may be surprised by the lack of an equivalent of the `return` command that would allow you to exit function with some return value. Instead in Opa always the *last expression* of the function is its return value.

This is a convention that Opa borrows from functional programming languages (as in fact Opa itself is, for the most part, functional!). It may feel limiting at first, but don't worry you will quickly get used to that and you may even start thinking of a disruption of the functions flow-of-control caused by `return` as almost as evil as that of the ill-famed `goto`...

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.

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.

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.

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
```

{block}[TIP] ### About comparison To compare two values, use operator `==` or, equivalently, function \==`(with the backquotes). When comparingx == y(or`==`(x,y)),xandymust have the same type. The result of a comparison is a boolean. We write that the type of function`==`is'a,'a -> bool`. {block}

{block}[TIP] ### About *booleans* In Opa, booleans are values {`true: void`} and {`false: void`}, or, more concisely but equivalently, {`true`} and {`false`}.

Their type declaration looks as follow: `type bool = {true} or {false}`. Such types, admitting one of a number of variants, are called sum types. {block}

{block}[TIP] ### About sum types A value has a *sum type* `t or u`, meaning that the values of this type are either of the two variants: either a value of type `t` or a value of type `u`.

A good example of sum type are the aforementioned boolean values, which are defined as `type bool = {false} or {true}`.

Another good example of sum type is the type `list` of linked lists; its definition can be summed up as `{nil} or {... hd, list tl}`.

Note that sum types are not limited to two cases. Sum types with tens of cases are not uncommon in real applications. {block}

Safely determining which variant was used to construct a value of a sum type can be accomplished with pattern matching.

{block}[TIP] ### 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 (EXPR) { case CASE_1: EXPR_1 case CASE_2: EXPR_2 default: EXPR_n }`

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`.

As an example, you can check whether boolean `b` is true or false by using `if b then ... else ...` or, equivalently, `match (b) { case {true}: ... case {false}: ... }` {block}

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.

// - Now, expand this beyond two icons. Of course, each user's icon should remain constant during the conversation.

User customization

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

{block}[CAUTION] ### 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. {block}

Security

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

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!

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.

Overview

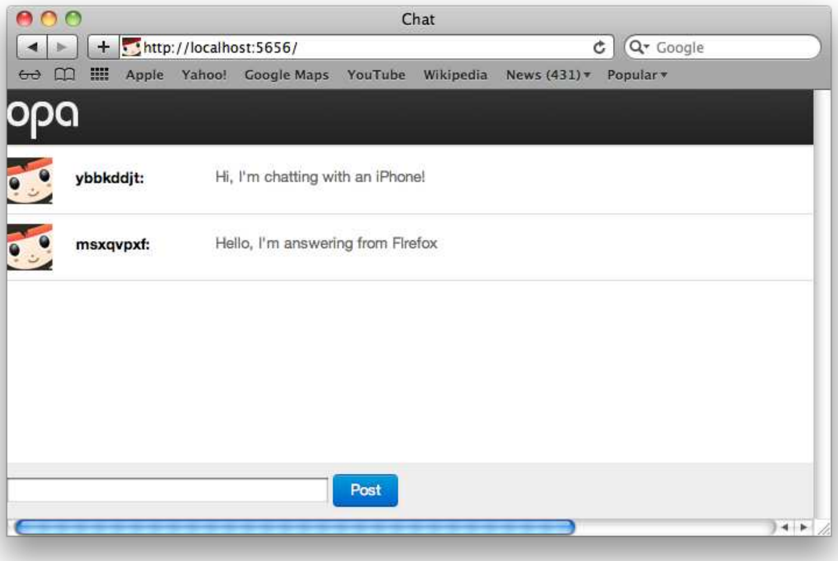
Let us start with a picture of the web chat we will develop in this chapter:

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.

Setting up communication

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:


Chat
http://localhost:5656/
Google
Apple Yahoo! Google Maps YouTube Wikipedia News (431) Popular
opa
Post

ybbkddjt: Hi, I'm chatting with an iPhone!

msxqvpxf: Hello, I'm answering from Firefox

Figure 4: Final version of the Hello chat application

`type message = {string author, string text}`[opa]

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

{block}[TIP] ### 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. {block}

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`[sh]

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. 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:

{block}[TIP] ### 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. {block}

`Network.network(message) room = Network.cloud("room")`[opa]

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.

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:

Skeleton of the user interface (incomplete)

```
<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.

{block}[TIP] ### 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 see later), apply to it functions accepting type `xhtml`, or use it as the body of a function. {block}

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

Skeleton of the user interface factorized as a function (still incomplete)

```
function 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` .

{block}[TIP] ### 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`.

The main syntax for defining a function is as follows: `function f(x1, x2, x3) { fun_body }`

Similarly, for a function with no argument, you can write

`function f() { fun_body }` 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()`. {block}

{block}[WARNING] ### Function syntax // FIXME: check In `function f(x1, x2, x3) { fun_body }` there is no space between `f` and `(`. Adding a space changes the meaning of the extract and would cause an error during compilation. {block}

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

Skeleton of the user interface with an arbitrary name (still incomplete)

```
function 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.

Sending and receiving

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

- at start-up, the application should *join* the room;
- whenever a message is broadcasted to the room, we should display it;
- 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.

Broadcasting a message to the room

```
function broadcast(author) {
    text = Dom.get_value(#entry);
    message = ~{author, text};
    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.

{block}[TIP] ### 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 variant of the selection operator is `#{id}`, which selects the DOM element whose id is the value of `id` (so `id` must be of type `string`). {block}

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: it can be abbreviated with `{~author, ~text}` or if all fields are constructed with such abbreviation even with: `~{author, text}`. So we could have written just as well:

Broadcasting a message to the room (variant)

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

{block}[TIP] ### 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. {block}

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:

Updating the user interface when a message is received

```
function user_update(message x) {
    line = <div>{x.author}: {x.text}</div>;
    #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 uses the special `#ID =+ HTML` construction to append the contents of `line` at the end of box with the id `conversation` that we have defined earlier. Instead of using the `=+` operator one can also use `+=` to *prepend* the element at the beginning of a given DOM node or simply `=` to *replace* the node with a given content.

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`.

{block}[TIP] ### 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. {block}

We are now ready to connect interactions.

Connecting interactions

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

Skeleton of the user interface connected to `broadcast` (still incomplete)

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

{block}[TIP] ### 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 online Opa API documentation by searching for entry `Dom.event`. {block}

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

Skeleton of the user interface now connected to everything (final version)

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

{block}[TIP] ### 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. {block}

Bundling, building, launching

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

{block}[TIP] ### 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. {block}

The server (first version)

```
Server.start(Server.http, {title: "Chat", page: start})
```

This extract launches a new HTTP server. For that the special `Server.start` construction is used.

{block}[TIP] ### About `Server.start` `Server.start` is, in a way, an equivalent of a `main` function in C/Java/..., as it's the entry point of the program. However, in Opa instead of just being a function to be executed when a program is started, `Server.start` starts an HTTP server to serve resources to the clients.

We will present some ways of defining servers below. You can also look up the `Server` module in online Opa API and learn more there. {block}

This is the type of the `Server.start` function:

`void Server.start(Server.conf configuration, Server.handler handler)`[opa]

It takes two parameters: configuration and a handler which essentialy defines a service.

Configuration, `Server.conf`, is just a simple record, which defines the port on which the server will run, it's netmask, used encryption and name. Most often you will use `Server.http` or `Server.https`; possibly customizing those values depending on your needs.

In this case the server is constructed using {`title: ..., page: ...`} record which builds a one-page server given a function `page` to generate this page and a `title`.

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

We have seen compilation already:

Compiling Hello, Chat

`opa hello_chat.opa`[sh]

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`[sh]

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

{block}[TIP] ### 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. {block}

{block}[TIP] ### 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. {block}

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

Perhaps it is time to add some style.

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] (https://developer.mozilla.org/En/CSS) at hand.

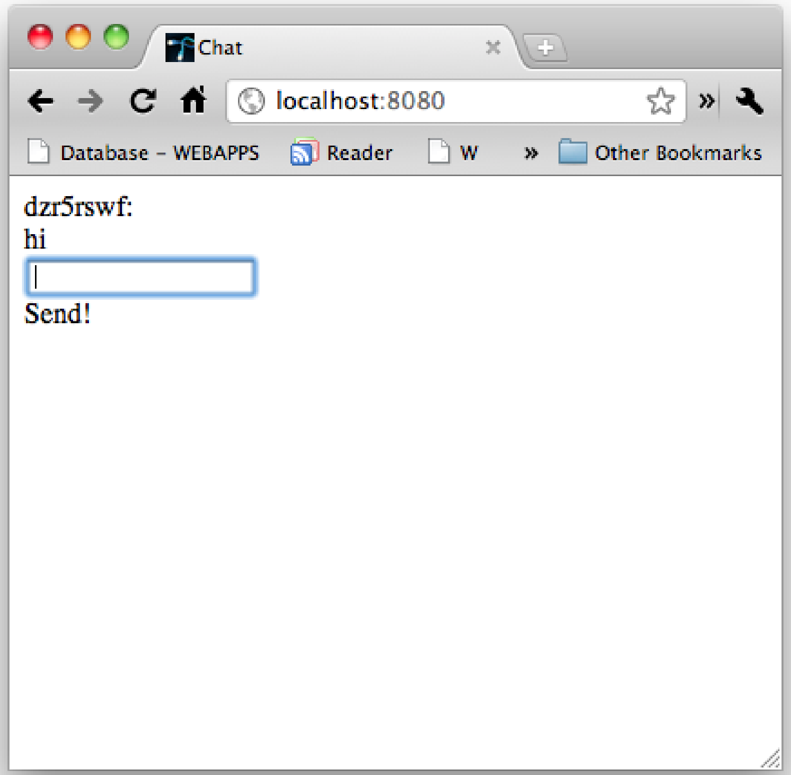
Of course, you will always need some custom CSS, specific to your application. Still, you can use some standard CSS to get you started with some predefined, nice-looking classes. Opa makes this as easy as a single import line:

`import stdlib.themes.bootstrap`[opa]

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):

Main user interface



dzr5rswf:
hi

Send!

Figure 5: Resulting application, missing style

```
<div class="topbar"><div class="fill"><div class="container"><div id=#logo /></div></div></d
<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:

Function to update the user interface when a message is received

```
function user_update(message x) {
    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>;
    #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:

Contents of file `resources/chat.css`

[css]file://hello_chat/resources/chat.css

Create a directory called `resources` and save this file as `resources/chat.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 extend our `Server.start` construct by instructing it to access the directory and to use our stylesheet. We can achieve by supplying `Server.start` with a record defining our server. More forms of records are allowed, see the definition of `Server.start` in the online API.

The server (final version)

```
Server.start(
    Server.http,
    [ { resources: @static_resource_directory("resources") }
```

```
      , { register: { css: ["resources/chat.css"] } }
      , { title: "Chat", page: start }
    ]
)
```

In this extract, we have instructed the Opa *compiler* to: * embed the files of directory `resources` and offer them to the browser * use a custom resource (CSS) located at `resources/chat.css` and * start a one-page application with title *Chat* and the content of the page generated by the +start+ function.

{block}[TIP] ### 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`. {block}

{block}[TIP] ### 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 `@`. {block}

While we are adding directives, let us take this opportunity to inform the compiler that the chatroom definition `room` should be visible and directly accessible for the clients.

`exposed @async room = Network.network(message)`[opa]

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:

The complete application

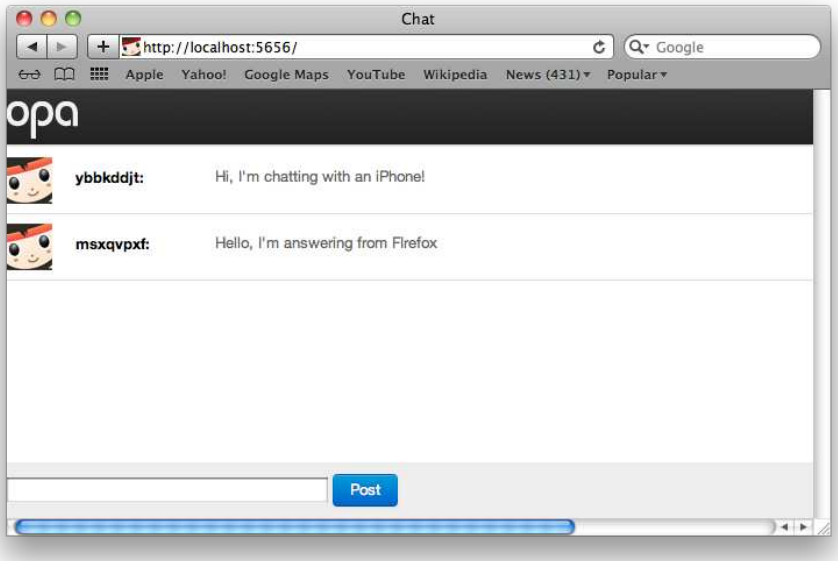
[opa|fork=hello_chat|run=http://chat.opalang.org]file://hello_chat/hello_chat.opa

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.

Questions

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?


Chat
http://localhost:5656/
Google
Apple Yahoo! Google Maps YouTube Wikipedia News (431) Popular
opa
Post

ybbkddjt: Hi, I'm chatting with an iPhone!

msxqvpxf: Hello, I'm answering from Firefox

Figure 6: Final version of the Hello chat application

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.

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.

Where is my `return`?

You may be surprised by the lack of an equivalent of the `return` command that would allow you to exit function with some return value. Instead in Opa always the *last expression* of the function is its return value.

This is a convention that Opa borrows from functional programming languages (as in fact Opa itself is, for the most part, functional!). It may feel limiting at first, but don't worry you will quickly get used to that and you may even start thinking of a disruption of the functions flow-of-control caused by `return` as almost as evil as that of the ill-famed `goto`...

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.

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.

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.

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
```

{block}[TIP] ### 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‘. {block}

{block}[TIP] ### About *booleans* In Opa, booleans are values {`true: void`} and {`false: void`}, or, more concisely but equivalently, {`true`} and {`false`}.

Their type declaration looks as follow: `type bool =` {`true`} `or` {`false`}. Such types, admitting one of a number of variants, are called sum types. {block}

{block}[TIP] ### About sum types A value has a *sum type* `t or u`, meaning that the values of this type are either of the two variants: either a value of type `t` or a value of type `u`.

A good example of sum type are the aforementioned boolean values, which are defined as `type bool =` {`false`} `or` {`true`}.

Another good example of sum type is the type `list` of linked lists; its definition can be summed up as {`nil`} `or` {`... hd, list tl`}.

Note that sum types are not limited to two cases. Sum types with tens of cases are not uncommon in real applications. {block}

Safely determining which variant was used to construct a value of a sum type can be accomplished with pattern matching.

{block}[TIP] ### 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 (EXPR)` { `case CASE_1: EXPR_1 case CASE_2: EXPR_2 default: EXPR_n` }

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`.

As an example, you can check whether boolean `b` is true or false by using `if b then ... else ...` or, equivalently, `match (b)` { `case` {`true`}: `... case` {`false`}: `...` } {block}

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.

// - Now, expand this beyond two icons. Of course, each user's icon should remain constant during the conversation.

User customization

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

{block}[CAUTION] ### 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. {block}

Security

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

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!

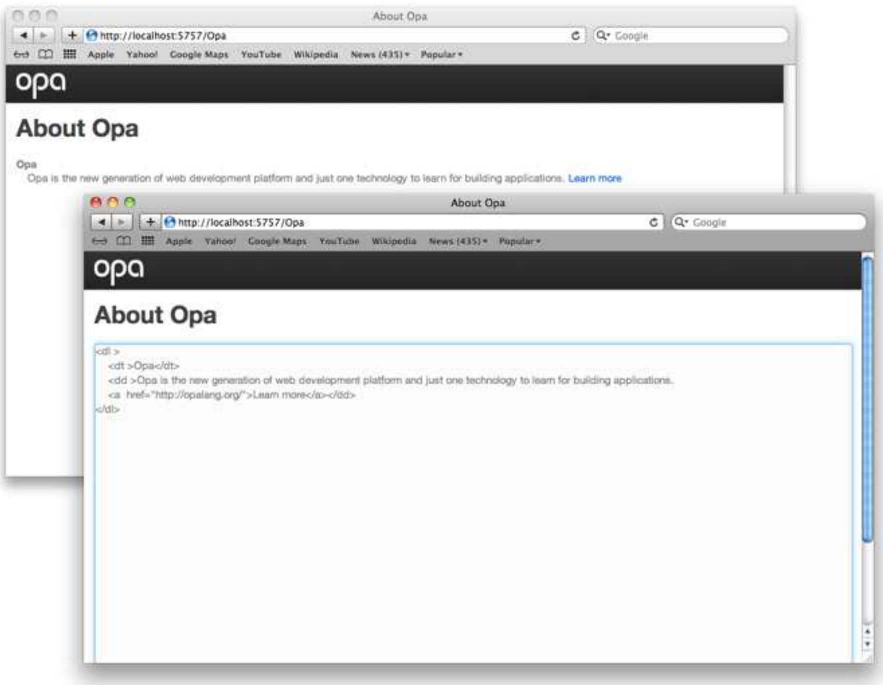
Hello, wiki

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.

Overview

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



Opa
Opa is the new generation of web development platform and just one technology to learn for building applications. Learn more

```
<dl >
   <dt >Opa</dt>
   <dd >Opa is the new generation of web development platform and just one technology to learn for building applications.
   <a  href="http://opalang.org/">Learn more</a></dd>
</dl>
```

Figure 7: Final version of the Hello wiki application

This web application stores pages and lets users edit them, using Markdown syntax, which is a popular markup language that supports headings, links, lists, images etc.

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

[opa|fork=hello_wiki|run=http://wiki.tutorials.opalang.org]file://hello_wiki/hello_wiki.opa

In this listing, we define a database for storing the content of the pages in the Markdown syntax 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.

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:

```
database stringmap(string) /wiki
```

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(string)`. This is a `stringmap` (i.e. an association from `string` keys to values) containing values of type `string` (used to denote Markdown code).

{block}[TIP] ### 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. {block}

{block}[TIP] ### HTML VS markup On the web, letting users directly write HTML code that can be seen at a later stage by other users is not a good idea, as it opens the way for numerous forms of attacks, sometimes quite subtle and hard to detect.

Opa offers at least two ways to circumvent this problem. Firstly, there is a *templating* mechanism (see package `stdlib.web.template`), which is an XML-based markup that covers a safe subset of HTML. Moreover this templating mechanism is designed for full extensibility – indeed, our website, including our own on-line editor, is developed using templating.

Another, more lightweight approach, and one that we are using here, is to use Markdown (package `stdlib.tools.markdown`), which is a very popular markup language that convers easy to read and write plain text format into structurally valid and completely safe XHTML (and supports headings, links, images, code snippets etc.). {block}

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

```
database /wiki[_] = "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 `"This page is empty.  Double-click to edit."`, i.e. a simple text in Markdown syntax (here it's just plain text).

With these two lines, the database is set. Any data written to the database will be kept persistent. Should you stop and restart your application, the data will be checked and made accessible to your application as it was at the point you stopped it.

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, `load_source`, will load some content from the database and present it as source code that the user can edit, while a second one, `load_rendered` will load the same content and present it as xhtml that can be displayed immediately:

```
function load_source(topic) {
  /wiki[topic]
}

function load_rendered(topic) {
  source = load_source(topic)
  Markdown.xhtml_of_string(Markdown.default_options, source)
}
```

In this extract, `topic` is the topic we wish to display or edit – i.e. the name of the page. 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 just return it as a `string`, or convert to `xhtml` data structure for display, using the `Markdown.xhtml_of_string` function, which parses the string and builds its corresponding `xhtml` representation, with respect to the Markdown syntax.

Saving data is equally simple:

```
function save_source(topic, source) {
    /wiki[topic] <- source;
    load_rendered(topic);
}
```

This function takes two arguments: a `topic`, with the same meaning as above, and a `source`, i.e. a `string`, which is a representation of the page with Markdown syntax. The instruction after `do` writes `source` to the database at path `/wiki[topic]`.

User interface

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

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

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 that will require different variant of `Server.start`, which we will see 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:

```
function edit(topic) {
    Dom.set_value(#edit_content, load_source(topic));
    Dom.hide(#show_content);
    Dom.show(#edit_content);
    Dom.give_focus(#edit_content);
}
```

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:

```
function save(topic) {
    content = save_source(topic, Dom.get_value(#edit_content));
    #show_content = content;
    Dom.hide(#edit_content);
    Dom.show(#show_content);
}
```

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

Serving the pages

For this application we will use a new variant of `Server.start`. Before we were using a construction for a single-page server; in practice most web applications have multiple pages. For that we can use `Server.start(Server.http, {dispatch: dispatch_fun})`, where `dispatch_fun` is a function that takes an `Uri.relative` and produces a `resource`.

Let's first construct such a function:

```
function start(url) {
  match (url) {
    case {path: {nil} ... } :
      { display("Hello") };
    case {path: path ...} :
      { display(String.concat("::", path)) };
  }
}
```

This is another pattern-matching, built from some constructions that you have not seen yet. Pattern `{path: [] ...}` accepts requests for `uri`s with empty `path`, such as "http://localhost:8080/". This is because ... 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.

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.

```
function start(url) {
  match (url) {
    case {path:[] ... } :
      { display("Hello") };
    case {~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`.

Adding some style

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

Contents of file `resources/css.css`

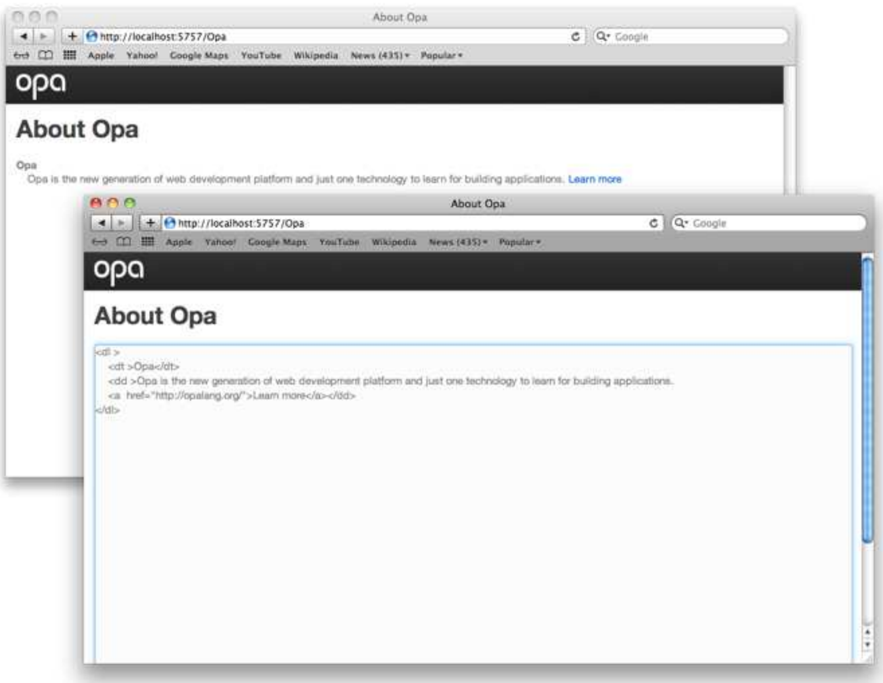
[css]file://hello_wiki/resources/css.css

One last step: including the stylesheet. For this purpose, we need to extend our server to also include resources. We do it as follows:

```
Server.start(Server.http,
   /** Statically embed a bundle of resources */
  [ {resources: @static_include_directory("resources")}
   /** Launch the [start] dispatcher */
  , {dispatch: start}
  ]
)
```

We provide `Server.start` here with a list of servers. Note that their order is important, as to serve a request they will be tried in that order. So we first put a resource bundle, which only handles requests to the defined resources and then we pass dispatching to our `start` function.

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



About Opa

Opa
Opa is the new generation of web development platform and just one technology to learn for building applications. Learn more

About Opa

```
<dl >
  <dt >Opa</dt>
  <dd >Opa is the new generation of web development platform and just one technology to learn for building applications.
  <a  href="http://opalang.org/">Learn more</a></dd>
</dl>
```

Figure 8: Final version of the Hello wiki application

As a summary, let us recapitulate the source file:

The complete application

[opa|fork=hello_wiki]file://hello_wiki/hello_wiki_simple.opa

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

Questions

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 one can never be interpreted as the other one.

{block}[CAUTION] ### 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. {block}

What about database security?

Now that we have a database, it is high time to think about *what* can be modified and *under what 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.

{block}[TIP] ### 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. {block}

Here, this means functions `edit` and `save`. In our listing, no other functions are *published*. Consequently, the `Markdown` syntax analysis is invoked only on the server.

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 special directive: `exposed`, which indicates that a given function shuld be exposed to the client.

{block}[WARNING] Plase note that from a security standpoint marking a function as `exposed` means that a compromised client can call such functions with arbitrary forged arugments; and not only those that would follow from the semantics of the application. {block}

To apply it to our three functions, we simply add it where needed:

```
exposed load_source(topic) { ... }
exposed load_rendered(topic) { ... }
exposed save_source(topic, source) { ... }
```

And we are done!

With this simple modification, saving now requires only one request. We will discuss `exposed` and the details of client-server *slicing* in a later chapter.

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

The complete application, made faster

[opa|fork=hello_wiki|run=http://wiki.tutorials.opalang.org]file://hello_wiki/hello_wiki.opa

Exercises

Time to put your new knowledge to the test.

Changing the default content

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

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?".

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.

Template chat

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

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.

{block}[TIP] ### 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} or {'a hd, list('a) tl}
```

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)
```

{block}

{block}[TIP] ### 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. {block}

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

Multi-room chat

Now that you know how to create multi-page servers, 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*.

{block}[TIP] ### 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`, which lets you force a function to be executed only on the server. {block}

{block}[TIP] ### 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. {block}

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!

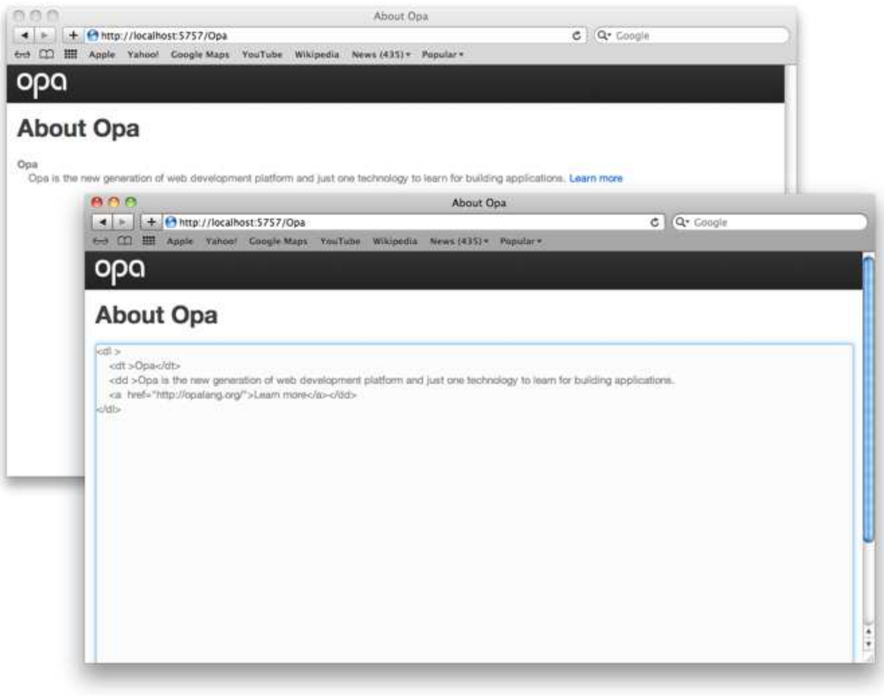
Hello, wiki

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.

Overview

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



About Opa

Opa
Opa is the new generation of web development platform and just one technology to learn for building applications. Learn more

About Opa

```
<dl >
   <dt >Opa</dt>
   <dd >Opa is the new generation of web development platform and just one technology to learn for building applications.
   <a  href="http://opalang.org/">Learn more</a></dd>
</dl>
```

Figure 9: Final version of the Hello wiki application

This web application stores pages and lets users edit them, using Markdown syntax, which is a popular markup language that supports headings, links, lists, images etc.

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

[opa|fork=hello_wiki|run=http://wiki.tutorials.opalang.org]file://hello_wiki/hello_wiki.opa

In this listing, we define a database for storing the content of the pages in the Markdown syntax 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.

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:

```
database stringmap(string) /wiki
```

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(string)`. This is a `stringmap` (i.e. an association from `string` keys to values) containing values of type `string` (used to denote Markdown code).

{block}[TIP] ### 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. {block}

{block}[TIP] ### HTML VS markup On the web, letting users directly write HTML code that can be seen at a later stage by other users is not a good idea, as it opens the way for numerous forms of attacks, sometimes quite subtle and hard to detect.

Opa offers at least two ways to circumvent this problem. Firstly, there is a *templating* mechanism (see package `stdlib.web.template`), which is an XML-based markup that covers a safe subset of HTML. Moreover this templating mechanism is designed for full extensibility – indeed, our website, including our own on-line editor, is developed using templating.

Another, more lightweight approach, and one that we are using here, is to use Markdown (package `stdlib.tools.markdown`), which is a very popular markup language that convers easy to read and write plain text format into structurally valid and completely safe XHTML (and supports headings, links, images, code snippets etc.). {block}

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

```
database /wiki[_] = "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 `"This page is empty.  Double-click to edit."`, i.e. a simple text in Markdown syntax (here it's just plain text).

With these two lines, the database is set. Any data written to the database will be kept persistent. Should you stop and restart your application, the data will be checked and made accessible to your application as it was at the point you stopped it.

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, `load_source`, will load some content from the database and present it as source code that the user can edit, while a second one, `load_rendered` will load the same content and present it as xhtml that can be displayed immediately:

```
function load_source(topic) {
  /wiki[topic]
}

function load_rendered(topic) {
  source = load_source(topic)
  Markdown.xhtml_of_string(Markdown.default_options, source)
}
```

In this extract, `topic` is the topic we wish to display or edit – i.e. the name of the page. 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 just return it as a `string`, or convert to `xhtml` data structure for display, using the `Markdown.xhtml_of_string` function, which parses the string and builds its corresponding `xhtml` representation, with respect to the Markdown syntax.

Saving data is equally simple:

```
function save_source(topic, source) {
    /wiki[topic] <- source;
    load_rendered(topic);
}
```

This function takes two arguments: a `topic`, with the same meaning as above, and a `source`, i.e. a `string`, which is a representation of the page with Markdown syntax. The instruction after `do` writes `source` to the database at path `/wiki[topic]`.

User interface

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

```
function display(topic) {
   xhtml =
     <div class="topbar"><div class="fill"><div class="container"><div id=#logo></div></div>
     <div class="content container">
```

```
        <div class="page-header"><h1>About {topic}</></>
        <div class="well" id=#show_content ondblclick={function(_) { edit(topic) }}>{load_ren
        <textarea rows="30" id=#edit_content onblur={function(_) { save(topic) }}></>
      </div>;
    Resource.styled_page("About {topic}", ["/resources/css.css"], xhtml);
}
```

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 that will require different variant of `Server.start`, which we will see 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:

```
function edit(topic) {
    Dom.set_value(#edit_content, load_source(topic));
    Dom.hide(#show_content);
    Dom.show(#edit_content);
    Dom.give_focus(#edit_content);
}
```

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:

```
function save(topic) {
    content = save_source(topic, Dom.get_value(#edit_content));
    #show_content = content;
    Dom.hide(#edit_content);
    Dom.show(#show_content);
}
```

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

Serving the pages

For this application we will use a new variant of `Server.start`. Before we were using a construction for a single-page server; in practice most web appliactions have multiple pages. For that we can use `Server.start(Server.http, {dispatch: dispatch_fun})`, where `dispatch_fun` is a function that takes an `Uri.relative` and produces a `resource`.

Let's first construct such a function:

```
function start(url) {
  match (url) {
    case {path: {nil} ... } :
      { display("Hello") };
    case {path: path ...} :
      { display(String.concat("::", path)) };
  }
}
```

This is another pattern-matching, built from some constructions that you have not seen yet. Pattern `{path: [] ...}` accepts requests for `uri`s with empty `path`, such as "http://localhost:8080/". This is because `...` 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.

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.

```
function start(url) {
  match (url) {
    case {path:[] ... } :
      { display("Hello") };
    case {~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`.

Adding some style

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

Contents of file `resources/css.css`

[css]file://hello_wiki/resources/css.css

One last step: including the stylesheet. For this purpose, we need to extend our server to also include resources. We do it as follows:

```
Server.start(Server.http,
   /** Statically embed a bundle of resources */
  [ {resources: @static_include_directory("resources")}
   /** Launch the [start] dispatcher */
  , {dispatch: start}
  ]
)
```

We provide `Server.start` here with a list of servers. Note that their order is important, as to serve a request they will be tried in that order. So we first put a resource bundle, which only handles requests to the defined resources and then we pass dispatching to our `start` function.

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:

The complete application

[opa|fork=hello_wiki]file://hello_wiki/hello_wiki_simple.opa

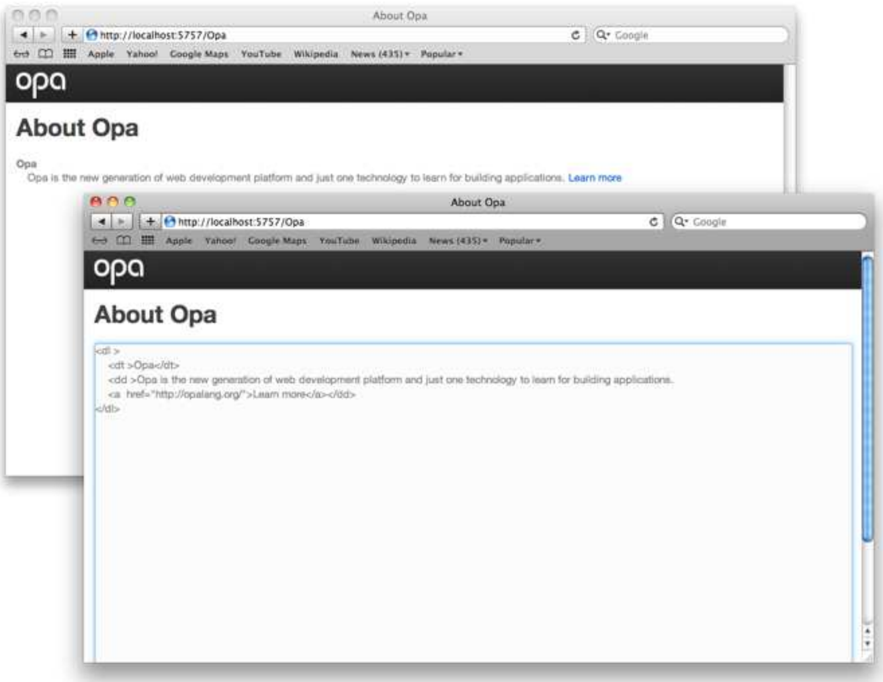
This is a total of 30 effective lines of code + CSS.

Questions

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



About Opa

Opa
Opa is the new generation of web development platform and just one technology to learn for building applications. Learn more

About Opa

```
<dl >
   <dt >Opa</dt>
   <dd >Opa is the new generation of web development platform and just one technology to learn for building applications.
   <a  href="http://opalang.org/">Learn more</a></dd>
</dl>
```

Figure 10: Final version of the Hello wiki application

difference between JavaScript code, text, or structured data, Opa does, and ensures that data that has been provided as one can never be interpreted as the other one.

{block}[CAUTION] ### 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. {block}

What about database security?

Now that we have a database, it is high time to think about *what* can be modified and *under what 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.

{block}[TIP] ### 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. {block}

Here, this means functions `edit` and `save`. In our listing, no other functions are *published.* Consequently, the `Markdown` syntax analysis is invoked only on the server.

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 special directive: `exposed`, which indicates that a given function shuld be exposed to the client.

{block}[WARNING] Plase note that from a security standpoint marking a function as `exposed` means that a compromised client can call such functions with arbitrary forged arugments; and not only those that would follow from the semantics of the application. {block}

To apply it to our three functions, we simply add it where needed:

```
exposed load_source(topic) { ... }
exposed load_rendered(topic) { ... }
exposed save_source(topic, source) { ... }
```

And we are done!

With this simple modification, saving now requires only one request. We will discuss `exposed` and the details of client-server *slicing* in a later chapter.

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

The complete application, made faster

[opa|fork=hello_wiki|run=http://wiki.tutorials.opalang.org]file://hello_wiki/hello_wiki.opa

Exercises

Time to put your new knowledge to the test.

Changing the default content

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

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?".

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.

Template chat

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

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.

{block}[TIP] ### 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} or {'a hd, list('a) tl}
```

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)
```

{block}

{block}[TIP] ### 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. {block}

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

Multi-room chat

Now that you know how to create multi-page servers, 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*.

{block}[TIP] ### 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`, which lets you force a function to be executed only on the server. {block}

{block}[TIP] ### 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. {block}

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!

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.

{block}[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*. {block}

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:

[opa|fork=hello_web_services|run=http://wiki-rest.tutorials.opalang.org]file://hello_web_services/hello_wiki_res

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

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:

```
function 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.

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 declared with `Server.start`:

The server, with an entry point for rest

```
function topic_of_path(path) {
    String.capitalize(String.to_lower(List.to_string_using("", "", "::", path)));
}

function start(url) {
    match (url) {
    case {path: [] ... }: display("Hello");
    case {path: ["_rest_" | path] ...}: rest(topic_of_path(path));
    case {~path ...}: display(topic_of_path(path));
    }
}

Server.start(Server.http,
  [ {bundle: @static_include_directory("resources")}
  , {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:

Handling rest requests

```
function rest(topic) {
  match (HttpRequest.get_method()) {
  case {some: method} :
      match (method) {
      case {post}:
          _ = save_source(
              topic,
              match (HttpRequest.get_body()) {
              case ~{some}: some;
              case {none}: "";
              }
          );
          Resource.raw_status({success});
      case {delete}:
          remove_topic(topic);
          Resource.raw_status({success});
      case {get}:
          Resource.raw_response(load_source(topic), "text/plain", {success});
      default:
          Resource.raw_status({method_not_allowed});
      }
    default:
      Resource.raw_status({bad_request});
  }
}
```

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:

Handling rest requests (shorter variant)

```
function rest(topic) {
    match (HttpServer.get_method()) {
    case {some : {post}} :
        _ = save_source(topic, HttpServer.get_body() ? "");
        Resource.raw_status({success});
    case {some : {delete}} :
        remove_topic(topic);
        Resource.raw_status({success});
    case {some : {get}} :
        Resource.raw_response(load_source(topic), "text/plain", {success});
    default :
        Resource.raw_status({method_not_allowed});
    }
}
```

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

[opa|fork=hello_web_services|run=http://wiki-rest.tutorials.opalang.org]file://hello_web_services/hello_wiki_res

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

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

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.

Questions

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.

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.

No need to worry, though, as we've already seen with Opa, you may combine any number of servers in an application. Take a look at all the variants in `Server.handler`, which is the type of the second argument to `Server.start`, to see other ways of constructing servers.

Exercises

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").

{block}[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. {block}

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

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

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`.

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.

{block}[TIP] ### 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. {block}

{block}[TIP] ### 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`). {block}

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.

{block}[TIP] ### JSON requests To obtain the JSON body of a request, use function `HttpRequest.get_json_body`. {block}

{block}[TIP] ### 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 }
 or { float Float }
 or { string String }
 or { bool Bool }
 or { list(RPC.Json.json) List }
 or { list((string, RPC.Json.json)) Record }
```

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

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.

{block}[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*. {block}

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:

[opa|fork=hello_web_services|run=http://wiki-rest.tutorials.opalang.org]file://hello_web_services/hello_wiki_res

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

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:

```
function 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.

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 declared with `Server.start`:

The server, with an entry point for rest

```
function topic_of_path(path) {
    String.capitalize(String.to_lower(List.to_string_using("", "", "::", path)));
}

function start(url) {
    match (url) {
    case {path: [] ... }: display("Hello");
    case {path: ["_rest_" | path] ...}: rest(topic_of_path(path));
    case {~path ...}: display(topic_of_path(path));
    }
}

Server.start(Server.http,
  [ {bundle: @static_include_directory("resources")}
  , {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:

Handling rest requests

```
function rest(topic) {
  match (HttpRequest.get_method()) {
  case {some: method} :
      match (method) {
      case {post}:
          _ = save_source(
              topic,
              match (HttpRequest.get_body()) {
              case ~{some}: some;
              case {none}: "";
              }
          );
          Resource.raw_status({success});
      case {delete}:
          remove_topic(topic);
          Resource.raw_status({success});
      case {get}:
          Resource.raw_response(load_source(topic), "text/plain", {success});
      default:
          Resource.raw_status({method_not_allowed});
      }
    default:
      Resource.raw_status({bad_request});
  }
}
```

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:

Handling rest requests (shorter variant)

```
function rest(topic) {
    match (HttpServer.get_method()) {
    case {some : {post}} :
        _ = save_source(topic, HttpServer.get_body() ? "");
        Resource.raw_status({success});
    case {some : {delete}} :
        remove_topic(topic);
        Resource.raw_status({success});
    case {some : {get}} :
        Resource.raw_response(load_source(topic), "text/plain", {success});
    default :
        Resource.raw_status({method_not_allowed});
    }
}
```

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

[opa|fork=hello_web_services|run=http://wiki-rest.tutorials.opalang.org]file://hello_web_services/hello_wiki_res

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

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

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.

Questions

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.

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.

No need to worry, though, as we've already seen with Opa, you may combine any number of servers in an application. Take a look at all the variants in `Server.handler`, which is the type of the second argument to `Server.start`, to see other ways of constructing servers.

Exercises

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").

{block}[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. {block}

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

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

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`.

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.

{block}[TIP] ### 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. {block}

{block}[TIP] ### 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`). {block}

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.

{block}[TIP] ### JSON requests To obtain the JSON body of a request, use function `HttpRequest.get_json_body`. {block}

{block}[TIP] ### 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 }
 or { float Float }
 or { string String }
 or { bool Bool }
 or { list(RPC.Json.json) List }
 or { list((string, RPC.Json.json)) Record }
```

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

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.

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):

[opa|fork=hello_web_services|run=http://wiki-rest-client.tutorials.opalang.org]file://hello_web_services/hello_w

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:

```
exposed function load_source(topic) {
    match (WebClient.Get.try_get(uri_for_topic(topic))) {
    case {failure: _} : "Error, could not connect";
    case {~success} :
        match (WebClient.Result.get_class(success)) {
        case {success}: success.content;
        default: "Error {success.code}";
        }
    }
}
```

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.) (`default` case). In case of success, we return the content of the response, e.g. `success.content`.

{block}[TIP] ### 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. {block}

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

```
function 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:

```
exposed function load_rendered(topic) {
    source = load_source(topic);
    Markdown.xhtml_of_string(Markdown.default_options, source);
}
```

Finally, we can adapt save_source, as follows:

```
exposed function save_source(topic, source) {
    match (WebClient.Post.try_post(uri_for_topic(topic), source)) {
    case { failure: _ }:
        {failure: "Could not reach the distant server"};
    case { success: s }:
        match (WebClient.Result.get_class(s)) {
        case {success}:
            {success: load_rendered(topic)};
        default:
            {failure: "Error {s.code}"};
        }
    }
}
```

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 either returns the result {success=...} or indicates an error with {failure=...}.

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

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:

```
function 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={function(_) { edit(topic) }}>{loa
    <div class="show_messages" id=#show_messages />
    <textarea class="edit_content" id=#edit_content style="display:none" cols="40" rows="30'
  );
}

function edit(topic) {
    #show_messages = <></>;
    Dom.set_value(#edit_content, load_source(topic));
    Dom.hide(#show_content);
    Dom.show(#edit_content);
    Dom.give_focus(#edit_content);
}

function save(topic) {
    match (save_source(topic, Dom.get_value(#edit_content))) {
    case { ~success }:
        #show_content = success;
        Dom.hide(#edit_content);
        Dom.show(#show_content);
    case {~failure}:
        #show_messages = <>{failure}</>;
    }
}
```

And that is all for the user interface.

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`.

{block}[TIP] ### 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 =
    { option(string)        schema
    , Uri.uri_credentials   credentials
    , string                domain
    , option(int)           port
    , list(string)          path
    , list((string,string)) query
    , option(string)        fragment
    }

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

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 or Uri.relative or ...
```

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`. {block}

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

`uri_for_topic` (first version)

```
function 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_", topic]
    , 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` (with derivation)

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

{block}[TIP]

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. {block}

With this, your client wiki is complete:

[opa|fork=hello_web_services|run=http://wiki-rest-client.tutorials.opalang.org]file://hello_web_services/hello_w

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?

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 =
  function(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. 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},
  }
  function (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*.

{block}[TIP] ### 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. {block}

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` with command-line filter (incomplete)

```
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: [],
    })
  function (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:

Parsing option `--wiki-server-port`

```
port_parser =
  {CommandLine.default_parser with
    names: ["--wiki-server-port"],
    description: "The server port of the REST server for this wiki. By default, 8080.",
    function on_param(x) { parser { case 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.default_parser`. 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`.

{block}[TIP] ### 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 {
case y=Rule.natural : //do something with y
case y=Rule.hex     : //do something with y
case "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`. {block}

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:

Parsing option `--wiki-server-domain`

```
domain_parser =
  {CommandLine.default_parser with
    names: ["--wiki-server-domain"],
    description: "The REST server for this wiki. By default, localhost.",
    function on_param(x) { parser { case 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:

Command-line arguments (complete)

```
uri_for_topic =
  domain_parser =
    {CommandLine.default_parser with
      names: ["--wiki-server-domain"],
      description: "The REST server for this wiki. By default, localhost.",
// FIXME, | after parser should not be needed
      function on_param(x) { parser { case 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.",
      function on_param(x) { parser { case y=Rule.natural -> {no_params: {x with port: {some
    }
  base_uri =
    CommandLine.filter(
      {title     : "Wiki arguments",
       init      : {Uri.default_absolute with domain: "localhost", schema: {some: "http"}},
       parsers   : [domain_parser, port_parser],
       anonymous : []
      }
    )
  function(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:

[opa|fork=hello_web_services|run=http://wiki-rest-client.tutorials.opalang.org]file://hello_web_services/hello_w

Exercises

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.

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.

{block}[TIP] ### 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`. {block}

Architecting a REST chat

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

{block}[TIP] ### 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. {block}

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.

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):

[opa|fork=hello_web_services|run=http://wiki-rest-client.tutorials.opalang.org]file://hello_web_services/hello_w

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:

```
exposed function load_source(topic) {
    match (WebClient.Get.try_get(uri_for_topic(topic))) {
    case {failure: _} : "Error, could not connect";
    case {~success} :
        match (WebClient.Result.get_class(success)) {
        case {success}: success.content;
        default: "Error {success.code}";
        }
    }
}
```

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.) (`default` case). In case of success, we return the content of the response, e.g. `success.content`.

{block}[TIP] ### 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. {block}

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

```
function 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:

```
exposed function load_rendered(topic) {
    source = load_source(topic);
    Markdown.xhtml_of_string(Markdown.default_options, source);
}
```

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

```
exposed function save_source(topic, source) {
    match (WebClient.Post.try_post(uri_for_topic(topic), source)) {
    case { failure: _ }:
        {failure: "Could not reach the distant server"};
    case { success: s }:
        match (WebClient.Result.get_class(s)) {
        case {success}:
            {success: load_rendered(topic)};
        default:
            {failure: "Error {s.code}"};
        }
    }
}
```

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 either returns the result {`success=...`} or indicates an error with {`failure=...`}.

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

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:

```
function 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={function(_) { edit(topic) }}>{loa
    <div class="show_messages" id=#show_messages />
    <textarea class="edit_content" id=#edit_content style="display:none" cols="40" rows="30"
  );
}

function edit(topic) {
    #show_messages = <></>;
    Dom.set_value(#edit_content, load_source(topic));
    Dom.hide(#show_content);
    Dom.show(#edit_content);
    Dom.give_focus(#edit_content);
}

function save(topic) {
    match (save_source(topic, Dom.get_value(#edit_content))) {
    case { ~success }:
        #show_content = success;
        Dom.hide(#edit_content);
        Dom.show(#show_content);
    case {~failure}:
        #show_messages = <>{failure}</>;
    }
}
```

And that is all for the user interface.

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`.

{block}[TIP] ### 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 =
    { option(string)        schema
    , Uri.uri_credentials   credentials
    , string                domain
    , option(int)           port
    , list(string)          path
    , list((string,string)) query
    , option(string)        fragment
    }

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

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 or Uri.relative or ...
```

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`. {block}

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

`uri_for_topic` (first version)

```
function 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_", topic]
    , 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` (with derivation)

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

{block}[TIP]

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. {block}

With this, your client wiki is complete:

[opa|fork=hello_web_services|run=http://wiki-rest-client.tutorials.opalang.org]file://hello_web_services/hello_w

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?

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 =
  function(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. 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},
  }
  function (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*.

{block}[TIP] ### 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. {block}

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` with command-line filter (incomplete)

```
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: [],
    })
  function (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:

Parsing option --wiki-server-port

```
port_parser =
  {CommandLine.default_parser with
    names: ["--wiki-server-port"],
    description: "The server port of the REST server for this wiki. By default, 8080.",
    function on_param(x) { parser { case 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.default_parser. 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.

{block}[TIP] ### 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 {
case y=Rule.natural : //do something with y
case y=Rule.hex     : //do something with y
case "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`. {block}

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:

Parsing option `--wiki-server-domain`

```
domain_parser =
  {CommandLine.default_parser with
    names: ["--wiki-server-domain"],
    description: "The REST server for this wiki. By default, localhost.",
    function on_param(x) { parser { case 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:

Command-line arguments (complete)

```
uri_for_topic =
  domain_parser =
    {CommandLine.default_parser with
      names: ["--wiki-server-domain"],
      description: "The REST server for this wiki. By default, localhost.",
// FIXME, | after parser should not be needed
      function on_param(x) { parser { case 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.",
      function on_param(x) { parser { case y=Rule.natural -> {no_params: {x with port: {some
    }
  base_uri =
    CommandLine.filter(
      {title     : "Wiki arguments",
       init      : {Uri.default_absolute with domain: "localhost", schema: {some: "http"}},
       parsers   : [domain_parser, port_parser],
       anonymous : []
      }
    )
  function(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:

[opa|fork=hello_web_services|run=http://wiki-rest-client.tutorials.opalang.org]file://hello_web_services/hello_w

Exercises

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.

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.

{block}[TIP] ### 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`. {block}

Architecting a REST chat

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

{block}[TIP] ### 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. {block}

Hello, database

In this chapter we will explore data storage possibilities. Opa has its own internal database, which is perfect for prototyping and quickly getting off the ground. However, for complex, data-intense applications, requiring scalable database with data replication and complex querying capabilities, Opa provides extensive support for the popular MongoDB database. This chapter focuses on MongoDB.

MongoDB: Quickstart

In this section we will take the simple counter example from the Opa tour chapter and run it using MongoDB.

First, unless done already, you need to download, install & run the MongoDB server. Installation essentially means unpacking the downloaded archive. And you can run the server with a command such as:

```
$ mkdir -p ~/mongodata/opa
$ {INSTALL_DIR}/mongod --noprealloc --dbpath ~/mongodata/opa > ~/mongodata/opa/log.txt 2>&1
```

which creates directory `~/mongodata/opa` for the data (1) and then runs MongoDB server using that location and writing logs to `~/mongodata/opa/log.txt` (2). The server will take a moment to start and once done you can verify that it works by running `mongo` (client) in another terminal. The response should look something like this (of course version number may vary):

```
$ {INSTALL_DIR}/mongo
MongoDB shell version: 2.0.2
connecting to: test
>
```

Let us recapitulate the example from the previous chapter (we will assume it is saved in a file `counter.opa`).

[opa|fork=hello-opa|run=http://hello-opa.tutorials.opalang.org]file://hello-opa/hello-opa.opa

So how do we upgrade this example to MongoDB? All you have to do is to use the compilation switch for setting the DB backend: `--database mongo` (MongoDB will be the default database in future releases of Opa and you will be able to omit the switch).

`opa --database mongo counter.opa`

run it as before:

```
./counter.exe
```

and... voilà, you have your first Opa application running on MongoDB!

{block}[WARNING] Beware, temporarily Opa accepts database declarations without explicit names, so:

```
database int /counter = 0;
```

instead of the above:

```
database mydb {
  int /counter = 0;
}
```

However, programs with such name-less databases will *not* work with MongoDB. This glitch will be removed in the next version of Opa. {block}

The above setup assumes that Mongo is running on local machine on its default port 27017; if that's not the case you can run your application (not the compiler!) with the option `--db-remote:db_name host[:port]`. For instance if Mongo was running locally but on port 4242, we'd need to run the application with:

```
./counter.exe --db-remote:mydb localhost:4242
```

Incidentally, if you use authentication with your MongoDB databases, you can provide the authentication parameters on the command line as follows:

```
./counter.exe --db-remote:mydb myusername:mypassword@localhost:4242
```

Note, however, that we have a slightly simplified view of MongoDB authentication in that we only support one database per connection. MongoDB implements authentication on a per-database basis and can have authentication to multiple databases over the same connection. If you have more than one database at the same server you wil have to issue multiple `--db-remote` options, one for each target database. A potentially useful feature is that if you authenticate with the `admin` database then it acts like ''root" access for databases and you can then access all databases over that connection.

So switching to Mongo is very easy. It is worth noting that although Mongo itself is "Schema free", the Opa compiler ensures adherence to program database definitions, therefore providing a safety layer and ensuring type safety for all database operations.

What are the gains of switching database backend? Apart from running on an industry standard DBMS it opens doors for more complex querying, which we will explore in the following section.

Database declarations

One example is worth a thousand words, so in the remainder of this section we will use a running example of a database for a movie rating service. First we need data for service users, which will just be a set of all users, each of them having her `id` of type `user_id` (here explicit `int`, though in a real-life example it would probably be abstracted away), an `int age` field and a `status` which is a simple variant type (enumeration).

```
type user_status = {regular} or {premium} or {admin}
type user_id = int
type user = { user_id id, string name, int age, user_status status }

database users {
  user /all[{id}]
  /all[_]/status = { regular }
}
```

First thing to notice is the notation for declaring *database sets*. Writing `user /user` declares a single value of type `user` at path `/user`, however `user /all[{id}]` declares a *set of* values of type `user`, where the record field `id` is the *primary key*.

Second important thing is that all database paths in Opa have associated *defaults values*. For `int` values that is `0`, for `string` values that is an empty string. For custom values we either need to provide it explicitly, which is what `/all[_]/status = { regular }` does, or we need to explicitly state that a given records will never be modified partially, which can be accomplished with the following declaration `/all[_] full` – more on partial record updates below.

Now we are ready to declare movies. First we introduce a type for a (movie) `person`, which in this simple example just consists of a `name` and the year of birth, `birthyear`.

```
type person = { string name, int birthyear }
```

Then we introduce a type for movie cast, including `director` and a list of `stars` playing in the movie (in credits order).

```
type cast = { person director, list(person) stars }
```

Now a (single) movie has its `title` (`string`), `view_counter` (`int`; how many times a given movie was looked at in the system), its `cast` and a set of ratings which are mappings from `int` (representing the rating) to `list(user_id)` (representing the list of users who gave that rating). It's not very realistic to just represent one single movie, but for our illustration purposes this will do.

```
database movie {
  string /title
  int /view_counter
  cast /cast
  intmap(list(user_id)) /ratings
}
```

It is worth noticing that we can store complex (non-primitive) values in the database, without any extra effort – `cast` is a record, which itself contains a list as one if its fields.

Here we notice that complex types – records (`cast`), `lists` (`stars` field of `cast`) and `intmap` – can be declared as entries in the database, in the same way as primitive tyes.

In the rest of this section we will explore how to read, query and update values of different types.

Basic types

We can read the title and the view counter of a single movie simply with:

```
string title = /movie/title;
int view_no = /movie/view_counter;
```

The `/movie/title` and `/movie/view_counter` are called *paths* and are composed of the database name (`movie`) and field names (`title`/`view_counter`).

Updates can be performed using the database write operator `path <- value`:

```
/movie/title <- "The Godfather";
/movie/view_counter <- 0;
```

What will happen if one tries to read unitialized value from the database? Say we do:

```
int view_no = /movie/view_counter;
```

when `/movie/view_counter` was never initialized; what's the value of `view_no`? We already mentioned default values above and the default value for a given path is exactly what will be given, if it was not written before.

This default value can be overwritten in the path declaration as follows:

```
database movies {
  ...
  int /view_counter = 10 // default value will be 10
}
```

Another option is to use question mark (`?`) before the path read, which will return an optional value, `{none}` indicating that the path has not been written to yet, as in:

```
match (?/movies/view_counter) {
case {none}: "No views for this movie...";
case {some: count}: "The movie has been viewed {count} times";
}
```

For integer fields we can also use some extra operators:

```
/movie/view_counter++;
/movie/view_counter += 5;
/movie/view_counter -= 10;
```

Records

With records we can do complete reads/updates in the same manner as for basic types:

```
cast complete_cast = /movie/cast;
/movie/cast <- { director: { name: "Francis Ford Coppola", birthyear: 1939 }
               , stars: [ { name: "Marlon Brando", birthyear: 1924 }, { name: "Al Pacino", b
               };
```

However, unless a given path has been declared with for full modifications only (`full` modificator explained above), we can also cross record boundaries and access/update only chosen fields, by including them in the path:

```
person director = /movie/cast/director;
/movie/cast/director/name <- "Francis Ford Coppola"
```

Also with updates we can only update some of the fields:

```
 // Notice the stars field below, which is not given and hence will not be updated
/movie/cast <- { director: { name: "Francis Ford Coppola", birthyear: 1939 } }
```

// TODO Explain defaults and need to define them for custom variant types

Lists

List in Opa are just (recursive) records and can be manipulated as such:

```
/movie/cast/stars <- []
person first_billed = /movie/cast/stars/hd
list(person) non_first_billed = /movie/cast/stars/tl
```

However, as it's a frequently used data-type, Opa provides a numer of extra operations on them:

```
 // removes first element of the list
/movie/cast/stars pop
 // removes last element of the list
/movie/cast/stars shift
 // appends one element to the list
/movie/cast/stars <+ { name: "James Caan", birthyear: 1940 }

person actor1 = ...
person actor2 = ...
 // appends several elements to the list
/movie/cast/stars <++ [ actor1, actor2 ]
```

Sets and Maps

Sets and maps are two types of collections that allow to organize multiple instances of data in the database. Sets represent a number of items of a given type, with no order between the items (as opposed to lists, which are ordered). Maps represent associations from keys to values.

We can fetch a single value from a given set by referencing it by its primary key:

```
user some_user = /users/all[{id: 123}]
```

Similarly for maps:

```
list(user_id) gave_one = /movie/ratings[1]
```

We can also make various queries using the following operators (*comma* separated).

- `field == expr`: value of `field` is *equal* to the expression `expr`.
- `field != expr`: value `field` is *not equal* to the expression `expr`.

- `field < expr`: value of `field` is *smaller than* that of `expr` (you can also use: `<=`, `>` and `>=` with).
- `field in list_expr`: value of `field` is *one of* those of the list `list_expr`.
- `q1 or q2`: satisfies query `q1` *or* `q2`.
- `q1 and q2`: satisfies queries `q1` *and* `q2`.
- `not q`: does not satisfy query `q`.
- `{f1 q1, f2 q2, ...}`: field `f1` satisfies `q1`, field `f2` satisfies `q2` etc.

and possibly some options (*semicolon* separated)

- `skip n`: skip the first `n` results (`n` an expression of type `int`)
- `limit n`: limit the result to the maximum of `n` results (`n` an expression of type `int`)
- `order fld (, fld)+`: order the results by given fields. Possible variants include: `fld`, `+fld` (sort ascending), `-fld` (sort descending), `fld=expr` (where `expr` needs to be of type `{up} or {down}`).

Note that if the result of a query is not fully constained by the primary key and hence can contain multiple values then the result of the query is of type dbset (for sets) or of the initial map type (for maps, giving a sub-map). You can manipulate a dbset with `DbSet.iterator` and Iter.

Examples for sets:

```
user /users/all[id == 123]  // accessing a single entry by primary key
dbset(user, _) john_does = /users/all[name == "John Doe"]  // return a set of values
it = DbSet.iterator(john_does) // then use Iter module

dbset(user, _) underages = /users/all[age < 18]
dbset(user, _) non_admins = /users/all[status in [{regular}, {premium}]]
dbset(user, _) /users/all[age >= 18 and status == {admin}]
dbset(user, _) /users/all[not status == {admin}]
 // showing second 50 results for users that are below 18 or above 62,
 // sorted by age (ascending) and then id (descending)
dbset(user, _) users1 = /users/all[age <= 18 or age >= 62; skip 50; limit 50; order +age, -i
```

Examples for maps:

```
 // users who rated the movie 10
list(user_id) loved_it = /movie/ratings[10]
 // users who rated the movie between 7 and 9 (inclusive)
list(user_id) liked_it = /movie/ratings[>= 7 and <= 9]
```

Hello, database

In this chapter we will explore data storage possibilities. Opa has its own internal database, which is perfect for prototyping and quickly getting off the ground. However, for complex, data-intense applications, requiring scalable database with data replication and complex querying capabilities, Opa provides extensive support for the popular MongoDB database. This chapter focuses on MongoDB.

MongoDB: Quickstart

In this section we will take the simple counter example from the Opa tour chapter and run it using MongoDB.

First, unless done already, you need to download, install & run the MongoDB server. Installation essentially means unpacking the downloaded archive. And you can run the server with a command such as:

```
$ mkdir -p ~/mongodata/opa
$ {INSTALL_DIR}/mongod --noprealloc --dbpath ~/mongodata/opa > ~/mongodata/opa/log.txt 2>&1
```

which creates directory `~/mongodata/opa` for the data (1) and then runs MongoDB server using that location and writing logs to `~/mongodata/opa/log.txt` (2). The server will take a moment to start and once done you can verify that it works by running `mongo` (client) in another terminal. The response should look something like this (of course version number may vary):

```
$ {INSTALL_DIR}/mongo
MongoDB shell version: 2.0.2
connecting to: test
>
```

Let us recapitulate the example from the previous chapter (we will assume it is saved in a file `counter.opa`).

[opa|fork=hello-opa|run=http://hello-opa.tutorials.opalang.org]file://hello-opa/hello-opa.opa

So how do we upgrade this example to MongoDB? All you have to do is to use the compilation switch for setting the DB backend: `--database mongo` (MongoDB will be the default database in future releases of Opa and you will be able to omit the switch).

`opa --database mongo counter.opa`

run it as before:

```
./counter.exe
```

and... voilà, you have your first Opa application running on MongoDB!

{block}[WARNING] Beware, temporarily Opa accepts database declarations without explicit names, so:

```
database int /counter = 0;
```

instead of the above:

```
database mydb {
  int /counter = 0;
}
```

However, programs with such name-less databases will *not* work with MongoDB. This glitch will be removed in the next version of Opa. {block}

The above setup assumes that Mongo is running on local machine on its default port 27017; if that's not the case you can run your application (not the compiler!) with the option `--db-remote:db_name host[:port]`. For instance if Mongo was running locally but on port 4242, we'd need to run the application with:

```
./counter.exe --db-remote:mydb localhost:4242
```

Incidentally, if you use authentication with your MongoDB databases, you can provide the authentication parameters on the command line as follows:

```
./counter.exe --db-remote:mydb myusername:mypassword@localhost:4242
```

Note, however, that we have a slightly simplified view of MongoDB authentication in that we only support one database per connection. MongoDB implements authentication on a per-database basis and can have authentication to multiple databases over the same connection. If you have more than one database at the same server you wil have to issue multiple `--db-remote` options, one for each target database. A potentially useful feature is that if you authenticate with the `admin` database then it acts like ''root" access for databases and you can then access all databases over that connection.

So switching to Mongo is very easy. It is worth noting that although Mongo itself is "Schema free", the Opa compiler ensures adherence to program database definitions, therefore providing a safety layer and ensuring type safety for all database operations.

What are the gains of switching database backend? Apart from running on an industry standard DBMS it opens doors for more complex querying, which we will explore in the following section.

Database declarations

One example is worth a thousand words, so in the remainder of this section we will use a running example of a database for a movie rating service. First we need data for service users, which will just be a set of all users, each of them having her `id` of type `user_id` (here explicit `int`, though in a real-life example it would probably be abstracted away), an `int age` field and a `status` which is a simple variant type (enumeration).

```
type user_status = {regular} or {premium} or {admin}
type user_id = int
type user = { user_id id, string name, int age, user_status status }

database users {
  user /all[{id}]
  /all[_]/status = { regular }
}
```

First thing to notice is the notation for declaring *database sets*. Writing `user /user` declares a single value of type `user` at path `/user`, however `user /all[{id}]` declares a *set of* values of type `user`, where the record field `id` is the *primary key*.

Second important thing is that all database paths in Opa have associated *defaults values*. For `int` values that is `0`, for `string` values that is an empty string. For custom values we either need to provide it explicitly, which is what `/all[_]/status = { regular }` does, or we need to explicitly state that a given records will never be modified partially, which can be accomplished with the following declaration `/all[_] full` – more on partial record updates below.

Now we are ready to declare movies. First we introduce a type for a (movie) `person`, which in this simple example just consists of a `name` and the year of birth, `birthyear`.

```
type person = { string name, int birthyear }
```

Then we introduce a type for movie cast, including `director` and a list of `stars` playing in the movie (in credits order).

```
type cast = { person director, list(person) stars }
```

Now a (single) movie has its `title` (`string`), `view_counter` (`int`; how many times a given movie was looked at in the system), its `cast` and a set of ratings which are mappings from `int` (representing the rating) to `list(user_id)` (representing the list of users who gave that rating). It's not very realistic to just represent one single movie, but for our illustration purposes this will do.

```
database movie {
  string /title
  int /view_counter
  cast /cast
  intmap(list(user_id)) /ratings
}
```

It is worth noticing that we can store complex (non-primitive) values in the database, without any extra effort – `cast` is a record, which itself contains a list as one if its fields.

Here we notice that complex types – records (`cast`), `lists` (`stars` field of `cast`) and `intmap` – can be declared as entries in the database, in the same way as primitive tyes.

In the rest of this section we will explore how to read, query and update values of different types.

Basic types

We can read the title and the view counter of a single movie simply with:

```
string title = /movie/title;
int view_no = /movie/view_counter;
```

The `/movie/title` and `/movie/view_counter` are called *paths* and are composed of the database name (`movie`) and field names (`title`/`view_counter`).

Updates can be performed using the database write operator `path <- value`:

```
/movie/title <- "The Godfather";
/movie/view_counter <- 0;
```

What will happen if one tries to read unitialized value from the database? Say we do:

```
int view_no = /movie/view_counter;
```

when `/movie/view_counter` was never initialized; what's the value of `view_no`? We already mentioned default values above and the default value for a given path is exactly what will be given, if it was not written before.

This default value can be overwritten in the path declaration as follows:

```
database movies {
  ...
  int /view_counter = 10 // default value will be 10
}
```

Another option is to use question mark (`?`) before the path read, which will return an optional value, {`none`} indicating that the path has not been written to yet, as in:

```
match (?/movies/view_counter) {
case {none}: "No views for this movie...";
case {some: count}: "The movie has been viewed {count} times";
}
```

For integer fields we can also use some extra operators:

```
/movie/view_counter++;
/movie/view_counter += 5;
/movie/view_counter -= 10;
```

Records

With records we can do complete reads/updates in the same manner as for basic types:

```
cast complete_cast = /movie/cast;
/movie/cast <- { director: { name: "Francis Ford Coppola", birthyear: 1939 }
               , stars: [ { name: "Marlon Brando", birthyear: 1924 }, { name: "Al Pacino", b
               };
```

However, unless a given path has been declared with for full modifications only (`full` modificator explained above), we can also cross record boundaries and access/update only chosen fields, by including them in the path:

```
person director = /movie/cast/director;
/movie/cast/director/name <- "Francis Ford Coppola"
```

Also with updates we can only update some of the fields:

```
 // Notice the stars field below, which is not given and hence will not be updated
/movie/cast <- { director: { name: "Francis Ford Coppola", birthyear: 1939 } }
```

// TODO Explain defaults and need to define them for custom variant types

Lists

List in Opa are just (recursive) records and can be manipulated as such:

```
/movie/cast/stars <- []
person first_billed = /movie/cast/stars/hd
list(person) non_first_billed = /movie/cast/stars/tl
```

However, as it's a frequently used data-type, Opa provides a numer of extra operations on them:

```
 // removes first element of the list
/movie/cast/stars pop
 // removes last element of the list
/movie/cast/stars shift
 // appends one element to the list
/movie/cast/stars <+ { name: "James Caan", birthyear: 1940 }

person actor1 = ...
person actor2 = ...
 // appends several elements to the list
/movie/cast/stars <++ [ actor1, actor2 ]
```

Sets and Maps

Sets and maps are two types of collections that allow to organize multiple instances of data in the database. Sets represent a number of items of a given type, with no order between the items (as opposed to lists, which are ordered). Maps represent associations from keys to values.

We can fetch a single value from a given set by referencing it by its primary key:

```
user some_user = /users/all[{id: 123}]
```

Similarly for maps:

```
list(user_id) gave_one = /movie/ratings[1]
```

We can also make various queries using the following operators (*comma* separated).

- `field == expr`: value of `field` is *equal* to the expression `expr`.
- `field != expr`: value `field` is *not equal* to the expression `expr`.

- `field < expr`: value of `field` is *smaller than* that of `expr` (you can also use: `<=`, `>` and `>=` with).
- `field in list_expr`: value of `field` is *one of* those of the list `list_expr`.
- `q1 or q2`: satisfies query `q1` *or* `q2`.
- `q1 and q2`: satisfies queries `q1` *and* `q2`.
- `not q`: does not satisfy query `q`.
- `{f1 q1, f2 q2, ...}`: field `f1` satisfies `q1`, field `f2` satisfies `q2` etc.

and possibly some options (*semicolon* separated)

- `skip n`: skip the first `n` results (`n` an expression of type `int`)
- `limit n`: limit the result to the maximum of `n` results (`n` an expression of type `int`)
- `order fld (, fld)+`: order the results by given fields. Possible variants include: `fld`, `+fld` (sort ascending), `-fld` (sort descending), `fld=expr` (where `expr` needs to be of type `{up} or {down}`).

Note that if the result of a query is not fully constained by the primary key and hence can contain multiple values then the result of the query is of type dbset (for sets) or of the initial map type (for maps, giving a sub-map). You can manipulate a dbset with `DbSet.iterator` and Iter.

Examples for sets:

```
user /users/all[id == 123]  // accessing a single entry by primary key
dbset(user, _) john_does = /users/all[name == "John Doe"]  // return a set of values
it = DbSet.iterator(john_does) // then use Iter module

dbset(user, _) underages = /users/all[age < 18]
dbset(user, _) non_admins = /users/all[status in [{regular}, {premium}]]
dbset(user, _) /users/all[age >= 18 and status == {admin}]
dbset(user, _) /users/all[not status == {admin}]
 // showing second 50 results for users that are below 18 or above 62,
 // sorted by age (ascending) and then id (descending)
dbset(user, _) users1 = /users/all[age <= 18 or age >= 62; skip 50; limit 50; order +age, -i
```

Examples for maps:

```
 // users who rated the movie 10
list(user_id) loved_it = /movie/ratings[10]
 // users who rated the movie between 7 and 9 (inclusive)
list(user_id) liked_it = /movie/ratings[>= 7 and <= 9]
```

Hello, scalability

// [WARNING] // .You may need to upgrade Opa // ============= // Examples in this chapter require a version of Opa dated from May the 25th, 2011, or later // i.e. build 28532 or greater. // =============

// // About this chapter: // Main author: ? // Paired author:? // // Topics: // - shared networks, shared sessions // - executing with distributed sessions // - distributed database // - executed with distributed database // - deploying a load-balancer // - creating a web service // - accessing a web service // - deploying on EC2 //

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.

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 !

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!

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.

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 --ha
```

Frequently Asked Questions

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:

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

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.

{block}[TIP] ### 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`. {block}

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`.

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.
$
```

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`.

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 the chapter Getting Opa.

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`.

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 lo
```

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

{block}[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. {block}

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
```

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.

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
```

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.

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
```

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
```

Hello, scalability

// [WARNING] // .You may need to upgrade Opa // ============= // Examples in this chapter require a version of Opa dated from May the 25th, 2011, or later // i.e. build 28532 or greater. // =============

// // About this chapter: // Main author: ? // Paired author:? // // Topics: // - shared networks, shared sessions // - executing with distributed sessions // - distributed database // - executed with distributed database // - deploying a load-balancer // - creating a web service // - accessing a web service // - deploying on EC2 //

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.

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 !

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!

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.

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 --hap
```

Frequently Asked Questions

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:

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

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.

{block}[TIP] ### 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`. {block}

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`.

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.
$
```

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`.

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 the chapter Getting Opa.

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`.

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 lo
```

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

{block}[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. {block}

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
```

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.

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
```

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.

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
```

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
```

//[[hello_recaptcha]] Hello, reCaptcha, and the rest of the world =============================

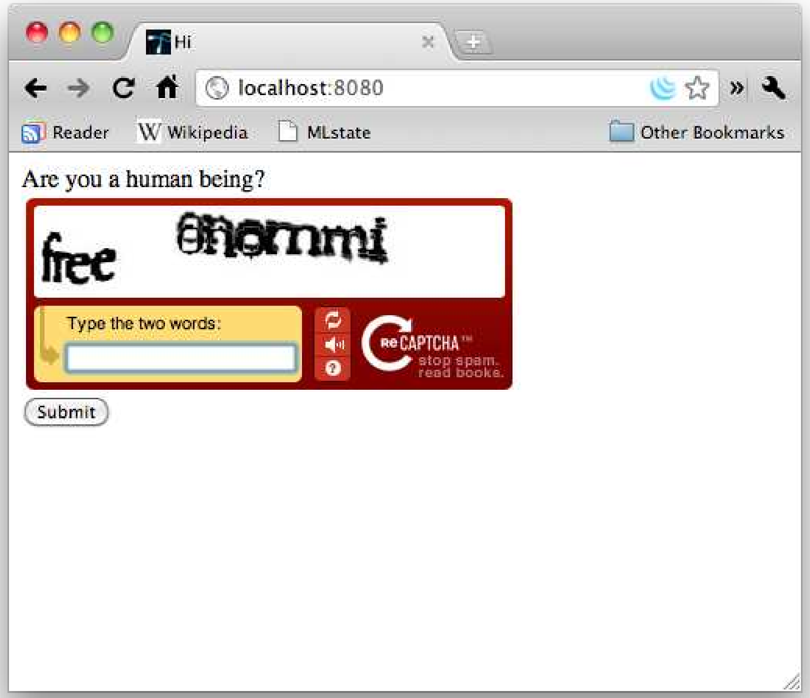
In this chapter, we will see how to plug an external API to the Opa platform – here, Google reCaptcha, an API used to protect forms against spammers by attempting to determine whether the person filling the form is actually a human being. Along the way, we will introduce the Opa Binding System Library (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.

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



Are you a human being?

Figure 11: Final version of the Hello reCaptcha application

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:

[opa|fork=hello_recaptcha|run=http://recaptcha.tutorials.opalang.org]file://hello_recaptcha/hello_recaptcha_ap

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:

[opa]file://hello_recaptcha/hello_recaptcha.opa

At low level, we find the JavaScript binding:

[js]file://hello_recaptcha/recaptcha.js

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

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.

{block}[TIP] ### 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. {block}

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:

Binding initialization (extract of `recaptcha.js`)

```
##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.

{block}[CAUTION] ### 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`. {block}

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:

Binding `reload`, `destroy` (extract of `recaptcha.js`)

```
##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.

Binding `get_challenge` and `get_response` (extract of `recaptcha.js`)

```
##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 `""`.

{block}[CAUTION] ### 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. {block}

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 <|opa_plugin_builder, its documentation|>.

We are now done with JavaScript.

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:

The reCaptcha configuration

```
type Recaptcha.config =
{
   {
     string privkey
   }
   cfg_private,

   {
      string pubkey,
      option(string) theme
   }
   cfg_public
}
```

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 `cfg_private`, for information that we want to keep on the server, and `cfg_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:

reCaptcha results

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

type Recaptcha.result = { Recaptcha.success success } or { Recaptcha.failure failure }
```

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

Type of the reCaptcha implementation (first version)

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

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.

{block}[TIP] ### 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*. {block}

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

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:

Function `onready` (first version)

```
function void onready(string id, string pubkey, string theme)
{
   Client.Script.load_uri_then(path_js_uri,
     function()
     {
       (%% 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.

{block}[TIP] ### 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 two `%%` is called the *key* of the external primitive. A part of the binding system reference details how keys are associated to primitives. {block}

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`.

{block}[TIP] ### 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. {block}

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

```
function validate(challenge, response, (Recaptcha.result -> void) callback)
{
  //By convention, do not even post a request if the data is empty
  if (String.is_empty(challenge) || String.is_empty(response))
  {
    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]
    function with_result(res)
    {
      match (res)
      {
        case ~{failure}:
          callback({failure: {captcha_not_reachable: failure}})
        case ~{success}:
          details = String.explode("\n", success.content)
          match (details)
          {
            case ["true" | _]: callback({success: {captcha_solved}})
            case ["false", code | _]: callback({failure: {upstream: code}})
            default: 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.implementation`, as follows:

```
function Recaptcha.implementation make_implementation(string privkey)
{
   function validate(challenge, response, (Recaptcha.result -> void) callback)
   {
     //By convention, do not even post a request if the data is empty
     if (String.is_empty(challenge) || String.is_empty(response))
     {
       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.resul
       function with_result(res)
       {
         match (res)
         {
           case ~{failure}:
             callback({failure: {captcha_not_reachable: failure}})
           case ~{success}:
             details = String.explode("\n", success.content)
             match (details)
             {
               case ["true" | _]: callback({success: {captcha_solved}})
               case ["false", code | _]: callback({failure: {upstream: code}})
               default: 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: dat
             with_result)
     }
```

```
    }
    {~validate, reload:cl_reload, destroy:cl_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:

```
function (Recaptcha.implementation, xhtml) make(Recaptcha.config config) {
    id = Dom.fresh_id();
    xhtml = <div id={id} onready={function(_) { onready(id, config.cfg_public.pubkey, config
    (make_implementation(config.cfg_private.privkey), xhtml);
}
```

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

```
function {string challenge, string response} get_token() {
    result = ({ challenge: (%%Recaptcha.get_challenge%%)()
              , response: (%%Recaptcha.get_response%%)()
              });
    (%%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.

Modularizing the features

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

Wrapping our code as a package

```
package tutorial.recaptcha
```

{block}[TIP] ### 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. {block}

Now that we have placed our code in a package, we may decide that some of our types are *abstract*, by adding an `@abstract` directive as follows:

```
abstract type Recaptcha.implementation = { ... }
```

{block}[TIP] ### 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. {block}

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:

Exporting features (first version)

```
function void validate(Recaptcha.implementation implementation, (Recaptcha.result -> void) c
    t = get_token();
    implementation.validate(t.challenge, t.response, callback);
}

function void reload(Recaptcha.implementation implementation) {
    implementation.reload();
}

function void destroy(Recaptcha.implementation implementation) {
    implementation.destroy();
}
```

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:

```
module Recaptcha
{
   function (Recaptcha.implementation, xhtml) make(Recaptcha.config config) {
       id = Dom.fresh_id();
       xhtml = <div id={id} onready={function(_) { onready(id, config.cfg_public.pubkey, con
       (make_implementation(config.cfg_private.privkey), xhtml);
   }

   function void validate(Recaptcha.implementation implementation, (Recaptcha.result -> voic
       t = get_token();
       implementation.validate(t.challenge, t.response, callback);
   }

   function void reload(Recaptcha.implementation implementation) {
       implementation.reload();
   }

   function void destroy(Recaptcha.implementation implementation) {
       implementation.destroy();
   }

   private path_validate_uri =
        Option.get(Parser.try_parse(Uri.uri_parser, "http://www.google.com/recaptcha/api/ver

   private path_js_uri =
        Option.get(Parser.try_parse(Uri.uri_parser, "http://www.google.com/recaptcha/api/js/

   private function void onready(string id, string pubkey, string theme) {
```

```
   Client.Script.load_uri_then(path_js_uri,
     function()
     {
       (%% Recaptcha.init %%)(id, pubkey, theme)
     }
    );
}

private cl_reload =
  %%Recaptcha.reload%% /**Implementation of [reload]*/

private cl_destroy =
  %%Recaptcha.destroy%% /**Implementation of [destroy]*/

private function Recaptcha.implementation make_implementation(string privkey) {
   function validate(challenge, response, (Recaptcha.result -> void) callback) {
     //By convention, do not even post a request if the data is empty
     if (String.is_empty(challenge) || String.is_empty(response)) {
       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.re
       function with_result(res) {
           match (res) {
           case ~{failure}:
               callback({failure: {captcha_not_reachable: failure}})
           case ~{success}:
               details = String.explode("\n", success.content)
               match (details) {
               case ["true" | _]: callback({success: {captcha_solved}})
               case ["false", code | _]: callback({failure: {upstream: code}})
               default: 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:
             with_result)
     }
   }
```

```
      {~validate, reload:cl_reload, destroy:cl_destroy}
   }

   private function {string challenge, string response} get_token() {
     result = ({ challenge: (%%Recaptcha.get_challenge%%)()
               , response: (%%Recaptcha.get_response%%)()
               });
     (%%Recaptcha.destroy%%)();
     result;
   }

}
```

{block}[TIP] ### About modules A *module* is an extended form of record introduced with `module ModuleName { ... }` 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. {block}

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:

[opa|fork=hello_recaptcha|run=http://recaptcha.tutorials.opalang.org]file://hello_recaptcha/hello_recaptcha.op

Testing the API

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

[opa]file://hello_recaptcha/hello_recaptcha_app.opa

With the exception of directives `server protected`, this listing should not surprise you. Here, we placed directives `server protected` as a sanity check, to be absolutely certain that the compiler would keep this value on the server and not expose it to the clients.

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
```

Questions

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.

{block}[TIP] ### 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. {block}

//[[hello_recaptcha]] Hello, reCaptcha, and the rest of the world ==============================

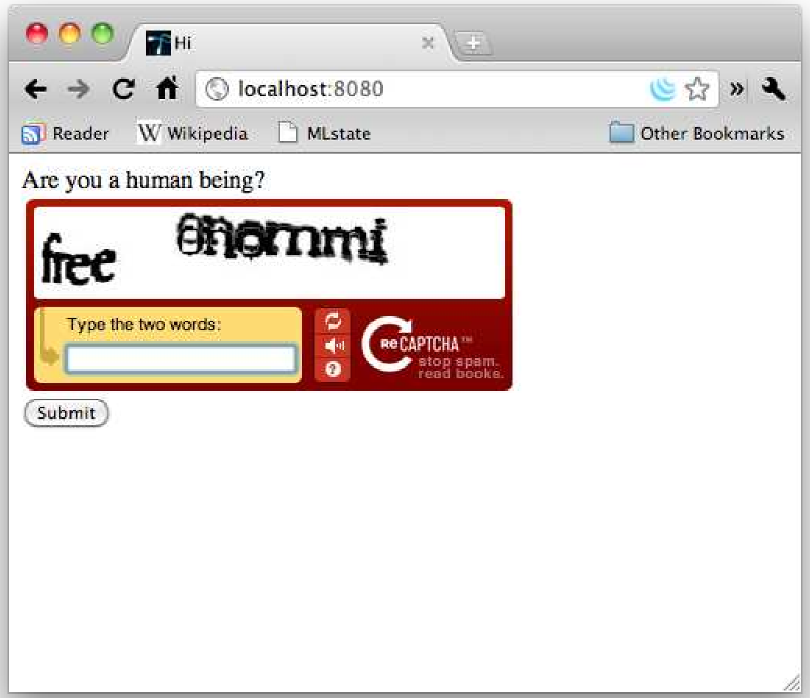
In this chapter, we will see how to plug an external API to the Opa platform – here, Google reCaptcha, an API used to protect forms against spammers by attempting to determine whether the person filling the form is actually a human being. Along the way, we will introduce the Opa Binding System Library (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.

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



Are you a human being?

Type the two words:

Submit

Figure 12: Final version of the Hello reCaptcha application

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:

[opa|fork=hello_recaptcha|run=http://recaptcha.tutorials.opalang.org]file://hello_recaptcha/hello_recaptcha_ap

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:

[opa]file://hello_recaptcha/hello_recaptcha.opa

At low level, we find the JavaScript binding:

[js]file://hello_recaptcha/recaptcha.js

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

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.

{block}[TIP] ### 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. {block}

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:

Binding initialization (extract of `recaptcha.js`)

```
##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.

{block}[CAUTION] ### 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`. {block}

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:

Binding `reload`, `destroy` (extract of `recaptcha.js`)

```
##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.

Binding `get_challenge` and `get_response` (extract of `recaptcha.js`)

```
##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 `""`.

{block}[CAUTION] ### 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. {block}

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 <|opa_plugin_builder, its documentation|>.

We are now done with JavaScript.

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:

The reCaptcha configuration

```
type Recaptcha.config =
{
   {
     string privkey
   }
   cfg_private,

   {
      string pubkey,
      option(string) theme
   }
   cfg_public
}
```

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 `cfg_private`, for information that we want to keep on the server, and `cfg_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:

reCaptcha results

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

type Recaptcha.result = { Recaptcha.success success } or { Recaptcha.failure failure }
```

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

Type of the reCaptcha implementation (first version)

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

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.

{block}[TIP] ### 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*. {block}

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

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:

Function `onready` (first version)

```
function void onready(string id, string pubkey, string theme)
{
   Client.Script.load_uri_then(path_js_uri,
     function()
```

```
    {
      (%% 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.

{block}[TIP] ### 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 two `%%` is called the *key* of the external primitive. A part of the binding system reference details how keys are associated to primitives. {block}

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`.

{block}[TIP] ### 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. {block}

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

```
function validate(challenge, response, (Recaptcha.result -> void) callback)
{
  //By convention, do not even post a request if the data is empty
  if (String.is_empty(challenge) || String.is_empty(response))
  {
    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]
    function with_result(res)
    {
      match (res)
      {
        case ~{failure}:
          callback({failure: {captcha_not_reachable: failure}})
        case ~{success}:
          details = String.explode("\n", success.content)
          match (details)
          {
            case ["true" | _]: callback({success: {captcha_solved}})
            case ["false", code | _]: callback({failure: {upstream: code}})
            default: 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.implementation`, as follows:

```
function Recaptcha.implementation make_implementation(string privkey)
{
   function validate(challenge, response, (Recaptcha.result -> void) callback)
   {
     //By convention, do not even post a request if the data is empty
     if (String.is_empty(challenge) || String.is_empty(response))
     {
       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.resul
       function with_result(res)
       {
         match (res)
         {
           case ~{failure}:
             callback({failure: {captcha_not_reachable: failure}})
           case ~{success}:
             details = String.explode("\n", success.content)
             match (details)
             {
               case ["true" | _]: callback({success: {captcha_solved}})
               case ["false", code | _]: callback({failure: {upstream: code}})
               default: 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: dat
             with_result)
     }
   }
   {~validate, reload:cl_reload, destroy:cl_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:

```
function (Recaptcha.implementation, xhtml) make(Recaptcha.config config) {
    id = Dom.fresh_id();
    xhtml = <div id={id} onready={function(_) { onready(id, config.cfg_public.pubkey, confi
    (make_implementation(config.cfg_private.privkey), xhtml);
}
```

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

```
function {string challenge, string response} get_token() {
    result = ({ challenge: (%%Recaptcha.get_challenge%%)()
                , response: (%%Recaptcha.get_response%%)()
              });
    (%%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.

Modularizing the features

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

Wrapping our code as a package

```
package tutorial.recaptcha
```

{block}[TIP] ### 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. {block}

Now that we have placed our code in a package, we may decide that some of our types are *abstract*, by adding an `@abstract` directive as follows:

```
abstract type Recaptcha.implementation = { ... }
```

{block}[TIP] ### 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. {block}

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:

Exporting features (first version)

```
function void validate(Recaptcha.implementation implementation, (Recaptcha.result -> void) c
    t = get_token();
    implementation.validate(t.challenge, t.response, callback);
}

function void reload(Recaptcha.implementation implementation) {
    implementation.reload();
}

function void destroy(Recaptcha.implementation implementation) {
    implementation.destroy();
}
```

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:

```
module Recaptcha
{
   function (Recaptcha.implementation, xhtml) make(Recaptcha.config config) {
       id = Dom.fresh_id();
       xhtml = <div id={id} onready={function(_) { onready(id, config.cfg_public.pubkey, con
       (make_implementation(config.cfg_private.privkey), xhtml);
   }

   function void validate(Recaptcha.implementation implementation, (Recaptcha.result -> voic
       t = get_token();
       implementation.validate(t.challenge, t.response, callback);
   }

   function void reload(Recaptcha.implementation implementation) {
       implementation.reload();
   }

   function void destroy(Recaptcha.implementation implementation) {
       implementation.destroy();
   }

   private path_validate_uri =
        Option.get(Parser.try_parse(Uri.uri_parser, "http://www.google.com/recaptcha/api/ver

   private path_js_uri =
        Option.get(Parser.try_parse(Uri.uri_parser, "http://www.google.com/recaptcha/api/js/

   private function void onready(string id, string pubkey, string theme) {
      Client.Script.load_uri_then(path_js_uri,
        function()
        {
```

```
      (%% Recaptcha.init %%)(id, pubkey, theme)
     }
    );
}

private cl_reload =
  %%Recaptcha.reload%% /**Implementation of [reload]*/

private cl_destroy =
  %%Recaptcha.destroy%% /**Implementation of [destroy]*/

private function Recaptcha.implementation make_implementation(string privkey) {
   function validate(challenge, response, (Recaptcha.result -> void) callback) {
     //By convention, do not even post a request if the data is empty
     if (String.is_empty(challenge) || String.is_empty(response)) {
       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.re
       function with_result(res) {
           match (res) {
           case ~{failure}:
               callback({failure: {captcha_not_reachable: failure}})
           case ~{success}:
               details = String.explode("\n", success.content)
               match (details) {
               case ["true" | _]: callback({success: {captcha_solved}})
               case ["false", code | _]: callback({failure: {upstream: code}})
               default: 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:
             with_result)
     }
   }
   {~validate, reload:cl_reload, destroy:cl_destroy}
}
```

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

}
```

{block}[TIP] ### About modules A *module* is an extended form of record introduced with `module ModuleName { ... }` 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. {block}

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:

[opa|fork=hello_recaptcha|run=http://recaptcha.tutorials.opalang.org]file://hello_recaptcha/hello_recaptcha.op

Testing the API

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

[opa]file://hello_recaptcha/hello_recaptcha_app.opa

With the exception of directives `server protected`, this listing should not surprise you. Here, we placed directives `server protected` as a sanity check, to be absolutely certain that the compiler would keep this value on the server and not expose it to the clients.

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
```

Questions

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.

{block}[TIP] ### 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. {block}

Hello, bindings – Binding 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, OCaml and Javascript, 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)

{block}[CAUTION] ### 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. {block}

Binding Ocaml and Javascript

This part shows a complete example, including compilation command line.

```
user@computer:~/$ ls
hello_binding.opa  plugin.js  plugin.ml  Makefile
```

Makefile file://opa_binding_examples/Makefile

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:

plugin.ml [ocaml]file://opa_binding_examples/plugin.ml

plugin.js [js]file://opa_binding_examples/plugin.js

```
##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.

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
```

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`).

[[key_normalization]] #### 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, they 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`

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 %%
```

{block}[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: {block}

```
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.

hello_bindings.opa [opa|fork=opa_bindings_examples]file://opa_binding_examples/hello_bindings.opa

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
```

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.

```
user@computer:~/$ ls
freq.opa  freq.c  c_binding.ml  Makefile
```

Makefile file://opa_binding_examples/c_binding/Makefile

freq.c [c]file://opa_binding_examples/c_binding/freq.c

c_binding.ml [ocaml]file://opa_binding_examples/c_binding/c_binding.ml

freq.opa [opa|fork=opa_bindings_examples]file://opa_binding_examples/c_binding/freq.opa

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.

Documentation

This paragraph describes the exhaustive list of directives handled by `opa-plugin-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.

{block}[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. {block}

General syntax

Syntax of types

```
type ::=
| int
| string
| float
| void
| bool
| option(<type>)
| <type>* sep , -> <type>
| <type-var>
| <type-ident> ( <type>* sep , )?
| opa[<type>]
| ( <type> )
```

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)

Syntax of arguments

```
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> )?
```

Example:

```
##extern-type hashset('a) = ('a, unit) Hashtbl.t
##extern-type black
```

Syntax of arguments for ##opa-type

```
arguments(##opa-type) ::=
| <type-ident> <parameters>?
```

Example:

```
##opa-type option('a)
```

Syntax of arguments for ##register

```
arguments(##register) ::=
| <name> : <type>
| <name> \ <name> : <type>
| <name> \ `<verbatim-ocaml>` : <type>
```

Example:

```
##register int_of_string : string -> option(int)
##register int_add \ ‘Pervasives.( + )‘ : int, int -> int
```

{block}[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. {block}

Example:

```
##register my_function :          \
   (int, int, int, int -> int),    \
   int ->                          \
   int
```

Tags Tags are needed for advanced uses only. The casual user can simply ignore them.

{table} {* 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 | - |} {table}

About Supported Types

{block}[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.) {block}

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.

{block}[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*) {block}

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
```

{block}[TIP] If the version of the compiler knows that some types have the same representation in Opa or in Ocaml, projections will not be generated. {block}

Representation of values

OCaml {table} {* 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}

Javascript {table} {* 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 |} {table}

An API is available for manipulating opa values, on the server side as well as on the client side. //<...>

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.

[[External_types]] ### 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.

{block}[TIP] In the standard library, low level arrays are implemented using extern-type definitions. {block}

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 file://opa_binding_examples/external_types/Makefile

bigint.ml file://opa_binding_examples/external_types/bigint.ml

bigint.opa file://opa_binding_examples/external_types/bigint.opa

[[Opa_types]] ### 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.

{block}[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. {block}

//[[client_lib]] //#### ClientLib //Coming soon… // TODO!

//[[server_lib]] //#### ServerLib //Coming soon… // TODO!

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.

{block}[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). {block}

```
user@computer:~/$ ls
dbm.ml  dbm.opa  Makefile
```

Makefile file://opa_binding_examples/opa_types/Makefile

dbm.ml file://opa_binding_examples/opa_types/dbm.ml

dbm.opa [opa|fork=opa_bindings_examples]file://opa_binding_examples/opa_types/dbm.opa

Exercise

Add a primitive which return a opa-list of entries, and add a button to show all entries of the database.

Hello, bindings – Binding 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, OCaml and Javascript, 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)

{block}[CAUTION] ### 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. {block}

Binding Ocaml and Javascript

This part shows a complete example, including compilation command line.

```
user@computer:~/$ ls
hello_binding.opa  plugin.js  plugin.ml  Makefile
```

Makefile file://opa_binding_examples/Makefile

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:

plugin.ml [ocaml]file://opa_binding_examples/plugin.ml

plugin.js [js]file://opa_binding_examples/plugin.js

```
##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.

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
```

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`).

[[key_normalization]] #### 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, they 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`

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 %%
```

{block}[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: {block}

```
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.

hello_bindings.opa [opa|fork=opa_bindings_examples]file://opa_binding_examples/hello_bindings.opa

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
```

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.

```
user@computer:~/$ ls
freq.opa  freq.c  c_binding.ml  Makefile
```

Makefile file://opa_binding_examples/c_binding/Makefile

freq.c [c]file://opa_binding_examples/c_binding/freq.c

c_binding.ml [ocaml]file://opa_binding_examples/c_binding/c_binding.ml

freq.opa [opa|fork=opa_bindings_examples]file://opa_binding_examples/c_binding/freq.opa

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.

Documentation

This paragraph describes the exhaustive list of directives handled by `opa-plugin-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.

{block}[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. {block}

General syntax

Syntax of types

```
type ::=
| int
| string
| float
| void
| bool
| option(<type>)
| <type>* sep , -> <type>
| <type-var>
| <type-ident> ( <type>* sep , )?
| opa[<type>]
| ( <type> )
```

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)

Syntax of arguments

```
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> )?
```

Example:

```
##extern-type hashset('a) = ('a, unit) Hashtbl.t
##extern-type black
```

Syntax of arguments for ##opa-type

```
arguments(##opa-type) ::=
| <type-ident> <parameters>?
```

Example:

```
##opa-type option('a)
```

Syntax of arguments for ##register

```
arguments(##register) ::=
| <name> : <type>
| <name> \ <name> : <type>
| <name> \ `<verbatim-ocaml>` : <type>
```

Example:

```
##register int_of_string : string -> option(int)
##register int_add \ `Pervasives.( + )` : int, int -> int
```

{block}[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. {block}

Example:

```
##register my_function :         \
   (int, int, int, int -> int),   \
   int ->                         \
   int
```

Tags Tags are needed for advanced uses only. The casual user can simply ignore them.

{table} {* 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 | - |} {table}

About Supported Types

{block}[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.) {block}

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.

{block}[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*) {block}

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
```

{block}[TIP] If the version of the compiler knows that some types have the same representation in Opa or in Ocaml, projections will not be generated. {block}

Representation of values

OCaml {table} {* 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}

Javascript {table} {* 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 |} {table}

An API is available for manipulating opa values, on the server side as well as on the client side. //<...>

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.

[[External_types]] ### 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.

{block}[TIP] In the standard library, low level arrays are implemented using extern-type definitions. {block}

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 file://opa_binding_examples/external_types/Makefile

bigint.ml file://opa_binding_examples/external_types/bigint.ml

bigint.opa file://opa_binding_examples/external_types/bigint.opa

[[Opa_types]] ### 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.

{block}[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. {block}

//[[client_lib]] //#### ClientLib //Coming soon... // TODO!

//[[server_lib]] //#### ServerLib //Coming soon... // TODO!

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.

{block}[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). {block}

```
user@computer:~/$ ls
dbm.ml  dbm.opa  Makefile
```

Makefile file://opa_binding_examples/opa_types/Makefile

dbm.ml file://opa_binding_examples/opa_types/dbm.ml

dbm.opa [opa|fork=opa_bindings_examples]file://opa_binding_examples/opa_types/dbm.opa

Exercise

Add a primitive which return a opa-list of entries, and add a button to show all entries of the database.

The core language

// // About this chapter: // Main author: ? // Paired author:? // // Topics: // - Full syntax // - The database // - Key functions of the standard library //

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.

Lexical conventions

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

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 `/**`.

Documentation

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

{block}[CAUTION] ###### In progress (Soon, a hyperlink to the corresponding chapter) {block}

// TODO TODO The syntax of document comment is described Ill-formed documentation comments do not break the compilation, they only break the documentation.

Basic datatypes

// no need to talk about char I think Opa has 3 basic datatypes: strings, integers and floating point numbers.

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
```

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
```

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"
function 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:

{table} {* 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 |} {table}

Datastructures

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 comma
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 b.c : "po"} //  means {x with b : {x.b 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}}
```

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")
@assert(x == {f1 : 1, f2 : "hello"});
@assert(x.f1 == 1);
@assert({x with f2 : "goodbye"} == (1,"goodbye"));
```

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:

```
list x = {nil}
list x = {hd:3, tl:{hd:4, tl:{hd:5, tl:{nil}}}}
list x = {hd:0, tl:{hd:1, tl:{hd:2, tl:x}}}
```

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:`function,module,with,type,recursive,and,match,if,as,case,default,else,database,parser,_,css,s

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: `end`, `external`, `forall`, `import`, `package`, `parser`, `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.

Bindings

At toplevel, you can define an identifier with the following syntax:

```
one = 1
‘hello‘ = "hello"
_z12 = 1+2
```

{block}[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 __. {block}

Of course, local identifiers can be defined too, and they are visible in the following expression:

```
two = {
  one = 1 // semicolon and newline are equivalent
  one + one
}

two = {
  one = 1; one + one // the exact same thing as above
}

two = {
  one = 1 // NOT VALID: syntax error because a local declaration
}         // must be followed by an expression
```

Functions

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:

```
function f(x,y){ // defining function f with the two parameters x and y
  x + y + 1
}

function int f(x,y){ // same as above but explicitly indicate the return type
  x + y + 1
}

two = {
  function f(x){ x + 1 } // functions be defined locally, just like other values
  f(1)
}

// you can write functions in a currified way concisely:
function f(x)(y){ x + y + 1 }
```

{block}[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.

```
function f (){ ... } // WARNING: parsed as an anonymous function which return a value of typ
x = f () // NOT VALID: parse error
```

{block}

Partial applications

From a function with N arguments, we may derive a function with less arguments by *partial application*:

```
function add(x,y){ x+y }
add1 = add(1,_) // which means function add1(y){ add(1,y) }
x = add1(2) // x is 3
```

{block}[CAUTION] Side effects of the arguments are computed at the site and time of partial application, not each time the function is called:

```
add1 = add(loop(), _) // this loops right now
                      // not when calling add1
```

{block}

[[underscore]] All the underscores of a call are regrouped to form the parameters of a unique function in the same order are the corresponding underscores:

```
function max3(x,y,z){ max(x,max(y,z)) }
positive_max = max3(0,_,_) // means function positive_max(x,y){ max(0,x,y) }
```

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:

```
function (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.

```
function{
case 0 : 1
case 1 : 2
case 2 : 3
default : 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).

```
function(e){
  match(e){
  case 0 : 1
  case 1 : 2
  case 2 : 3
  default : error("Wow, that's outside my capabilities")
  }
}
```

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.

{table} {* leading characters | associativity *} {| | @ | left |} {| || ? | right |} {| & | right |} {| = != > < | left |} {| + - ^ | left |} {| * / | left |} {table}

{block}[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
```

{block}

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:

```
list(int) x = [] // same as s = list(int) []
function list(int) f(x){ [x] } // annotation of the body of the function
                               // same as function f(x){ list(int) [x] }
```

Grouping

Expressions can be grouped with parentheses:

```
x = (1 + 2) * 3
```

Modules

Functionalities are usually regrouped into modules.

```
module List{
  empty = []
  function 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 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:

```
module 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):

```
module M{
  x = y
  y = 1
}
```

This will work just fine, because this is reordered into:

```
module M{
  y = 1
  x = y
}
```

where you have no more recursion.

{block}[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. {block}

Sequencing computations

In Opa the toplevel is executed, and so you can have expressions at the toplevel:

```
println("Executed!")
```

In a block if an expression is not binded and if not the last expression, this expression is computed and the result is discarded.

```
x = {
  println("Dibbs!"); // cleaner than saying _unused_name = println("Dibbs!")
                     // but equivalent (almost, see the warning section)
  println("Aww...");
  1
}
```

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}
@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 = bool {true}
@assert(x.true); // NOT VALID: type error
```

To deconstruct both records and sums, Opa offers *pattern-matching.* The general syntax is:

```
  match(<expr>){
  case <pattern_1> : <expr_1>
  case <pattern_2> : <expr_2>
  ...
  case <pattern_n> : <expr_n>
  default : <expr_default>
  }
```

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 <`expr_default`>. Note the default case (or equivalent `case _`) can be omitted.

The specific case of pattern matching on boolean can be abreviated using a standard `if-then-else` construct:

```
if(1 == 2){
  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
if(1 == 2) println("Who would have known that 1 == 2?")

// or equivalently
match(1 == 2){
case {true} : println("Who would have known that 1 == 2?")
case {false} : println("That's what I thought!")
}
```

{block}[TIP] The same way that `f(x,_)` means (roughly) `function(y){ f(x,y) }`, `_.label` is a shorthand for `function(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]
```

{block}

[[pattern]] ### Patterns

Generally, patterns appear as part of a `match` 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>){case <pattern> : <expr2> case ... }` is executed, `<expr>` is evaluated to a value, which is then matched against each pattern in order until a match is found.

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.

{block}[CAUTION] ###### Pattern-matching does not test for equality Consider the following extract:

```
x = 1
y = 2
match(y){
case x : println("Hey, 1=2")
default : 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
match(y){
case z : println("Hey, 1=2")
default : 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. {block}

A few examples:

```
function list_is_empty(l){
  match(l){
  case [] : true
  case [_|_] : false
  }
}

// and without the syntactic sugar for lists
// a list is either {nil} or {hd tl}
function head(l){
  match(list l){
  case {nil} : @fail
  case ~{hd ...} : hd
  }
}
```

{block}[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. {block}

{block}[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. {block}

{block}[WARNING] A pattern cannot contain an expression:

```
function is_zero(x){ // works fine
  match(x){
  case 0 : true
  case _ : false
  }
}

// wrong example
zero = 0
function is_zero(x){
  match(x){
  case zero : true
  case _ : false
  }
}
// does not work because the pattern defines zero
// it does not check that the x is equal to zero
```

{block}

{block}[CAUTION] You cannot put the same variable several times in the same pattern:

```
function on_the_diagonal(position){
  match(position){
  case {x=value y=value} : true
  case _ : false
}
// this is not valid because you are trying to give the name value
// to two values

// this must be written
function on_the_diagonal(position){
  position.x == position.y
}
```

{block}

//Loops //—– // //TODO Rudy - Where are we about 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 .

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{
    case "-"  [0-9]+ : {negative}
    case "+"? [0-9]+ : {positive}
  }
```

A parser is composed of a disjunction of `case <list-of-subrules> (: <semantic-action>)?. When the semantic action is omitted, it defaults to the`text‘ that was parsed by the left hand side.

A subrule consists of: - an optional binder that names the result of the subrule. It can be: - `x=` to name the result `x` - `~` 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*` - an optional prefix modifier (`!` or `&`) that lookahead for the following subrule in the input - a basic subrule - 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
```

{block}[CAUTION] Putting parentheses around a parser can change the type of the parenthesized parsers:

```
parser{
  case x=.* : ... // x as type list(Unicode.character)
}
parser{
  case 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. {block}

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{
  case "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"`.

Recursion

By default, toplevel functions and modules are implicitely recursive at toplevel, while local values (including values defined in functions) are not.

```
function f(){ f() } // an infinite loop

x =
  function f(){ f() } // NOT VALID: f is unbound
  void

x =
  recursive function f(){ f() } // now f is visible in its body
  void

function f(){ g() } // mutual recursion works without having
function g(){ f() }// to say 'recursive' anywhere at toplevel

x =
  recursive function f(){ g() } // local mutually recursive functions must
  and function g(){ f() } // be defined together with a 'recursive' '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:

```
module M{
  function f(){ M2.f() }
}
module M2{
  function f(){ m.f() }
}
```

This is valid, because it amounts to:

```
recursive function M_f(){ M2_f() }
and function 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:

```
recursive sess = Session.make(callback)
and function callback(){ /*do something with sess*/ }
```

Please note that the word `recursive` is meant to define *recursive* values, but not meant to define *cyclic* values:

```
recursive x = [x]
```

This definition is invalid, and will be rejected (statically in this case despite the presence of the `recursive` because it is sure to fail at runtime).

Of course, most invalid definitions will be only detected at runtime:

```
recursive x = if(true){ x } else { 0 }
```

Directives

Many special behaviours appear syntactically as directives. - A directive starting with a `@` - Expect the most common directives which are "both" / "server" / "client" / "exposed" / "protected" / "private" / "abstract" 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.

```
if true then void else @fail // @fail appears only in expressions
@expand function ‘=>‘(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. * `@stringifier`:: Takes a typename Overrides the generic stringification for the given type with the function annotated:

```
@stringifier(bool) function to_string(b: bool){ if(b){ "true" } else { "false" } }
```

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
```

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.*
```

{block}[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. {block}

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.

Type expressions

Type expressions are used in type annotations, and in type definitions.

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.

Record types

The syntax for record type works the same as it does in expressions and in patterns:

```
{useless} x = @fail // means {useless:void}
~{a, b} x = @fail // means {a a, b b}, where a and b are typenames
~{list} x = @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)
{a, b, ...} x = {a, b, c} // you can give only a part of the fields in type annotations
```

Tuple types

The type of a tuple actually looks like the tuple:

```
(int,float) (a,b) = (1,3.4)
```

Sum types

Now, record expressions do not have records type (in general), they have sum types, which are simply unions of record types:

```
({true} or {false}) x = {true}
({true} or ...) x = {true} // sum types can be partially specified, just like record types
```

Type names

Types can be given names, and of course you can refer to names in expressions:

```
list(int) x = [1] // the parameters of a type are written just like a function call
bool x = 1 // except that when there is no parameter, you don't write empty parentheses
list x = [1] // and except that you can omit all the parameters of a typename altogether
               // (which means 'please fill up with fresh variables for me')
```

Variables

Variables begin with an apostrophe except _:

```
list('a) x = []
list(_) x = [] // _ is an anonymous variable
```

Function types

Function types are list of types arguments separated by comma then a right arrow precedes type of result:

```
(int, int, int -> int) function max3(x, y, z){
  max(x, max(y, z))
}
```

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} or {false} // naming a type sum
type option('a) = {none} or {'a some} // a parameterized definition of a type sum
type list('a) = {nil} or {'a hd, list('a) tl} // 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
```

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
module Test1{
  Test1.t x = 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 Test1.t = int
module Test1{
  Test1.t x = 1 // will not compile since the module exports
                // Test1.x that has the private type t
}
```

Formal description

This syntax recapitulates the syntactic constructs of the language.

//[|grammar_conventions|] ### 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>
```

The opa language

A source file is a <`program`>, defined as follows:

```
program ::= <declaration>* sep <separator>
declaration ::=
  | <package-declaration>
  | <package-import>
  | <type-definition>
  | <binding>
  | <expr>
```

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>

separator ::=
  | ;
  | \n
```

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 ::=
  | int
  | string
  | float
  | <type-ident> ( <type>* sep , )
  | <record-type>
  | <lambda-type>
  | <sum-type>
  | forall ( <type-var>* sep , ) . <type>
  | <type-var>
  | <tuple-type>
  | ( <type> )

type-ident ::= <long-ident>

record-type ::=
  | ~? { <record-type-field>* sep , <record-type-end>? }
record-type-field ::=
  | <type> <field>
  | ~ <field>
  | <field>
record-type-end ::=
  | , <record-type-var>?
record-type-var ::=
  | ...
  | ' <ident>

lambda-type ::=
  | <type>* sep , -> <type>

sum-type ::=
  | or? <type> or <type>+ sep or <sum-type-end>?
  | <type> or? <sum-type-var>
sum-type-end ::=
  | or <sum-type-var>?
sum-type-var ::=
  | ...
  | ' <ident>

type-var ::=
  | ' <ident>
  | _

tuple-type ::=
  | ( <type> , )
  | ( <type> , <type>+ sep , ,? )
```

The syntax of the patterns:

```
pattern ::=
  | <literal>
  | <record-pattern>
  | <list-pattern>
  | <tuple-pattern>
  | <pattern> as <ident>
  | <pattern> | <pattern>
  | <type> <pattern>
  | ( <pattern> )
  | _

record-pattern ::=
  | ~? { ...? }
  | ~? { <record-pattern-field>+ sep , ,? ...? }
record-pattern-field ::=
  | ~? <type>? <field>
  | ~? <type>? <field> : <pattern>

list-pattern ::=
 | [ pattern+ sep , | pattern ]
 | [ pattern* sep , ,?]

tuple-pattern ::=
  | ( <pattern> , )
  | ( <pattern> , <pattern>+ sep , ,? )
```

The syntax of the bindings:

```
binding ::=
  | <binding-directive> recursive <non-rec-binding>+ sep and
  | <binding-directive> <non-rec-binding>

non-rec-binding ::=
  | <ident-binding>
  | <pattern-binding>

ident-binding ::=
  | <type>? <ident> = <expr>
  | <type>? function <type>? <ident> <params>+ { <expr_block> }
  | <type>? module <ident> <params>+ { <non-rec-binding>* sep <separator> }

params ::=
  | ( <pattern>* sep , )

pattern-binding ::=
  | <pattern> = <expr>
```

The syntax of the expressions, except the parsers: // todo database, domaction, xhtml. . .

```
expr ::=
  | <ident>
  | <expr-or-underscore> <op> <expr-or-underscore>
  | - <expr>
  | <type> <expr>
  | <expr-or-underscore> . field
  | <expr-or-underscore> ( <expr-or-underscore>* sep , )
  | <binding> <separator> <expr>
  | <match>
  | <lambda>
  | <module>
  | <record>
  | { <expr_block> }
  | ( expr )
  | <tuple>
  | <list>
  | <literal>
  | <directive>
  | <sysbinding>
  | <parser>

expr_block ::=
  | <expr>
  | <expr_or_binding>+ sep <separator> <separator> <expr>
expr_or_binding ::=
  | <expr>
  | <binding>

match ::=
  | match ( <expr> ){ <match_case>* <match_default> }
  | if <expr> then <expr> else <expr>
  | if <expr> then <expr>
match_case ::=
  | case <pattern> : <expr_block>
  | default : <expr_block>

lambda ::=
  | function <type>? <space> <params>+ { <expr_block> }
  | function{ <match_case>* <match_default> }

record ::=
  | ~? { <record-field>* sep ,? }
  | ~? { <expr> with <record-field>+ sep ;? }
```

```
record-field ::=
  | ~? <type>? <field>
  | <type>? <field> : <expr>

tuple ::=
  | ( <expr> , )
  | ( <expr> , <expr>+ sep , ,? )

list ::=
  | [ <expr>+ sep , | <expr> ]
  | [ <expr>* sep , ,? ]

module ::=
  | module{ <non-rec-binding>+ sep <separator> }

expr-or-underscore ::=
  | <expr>
  | _
```

The syntax of the parsers:

```
parser ::=
  | parser{ <parser-case>+ }
parser-case ::=
 | case <parser-rule>
parser-rule ::=
  | <parser-prod>+ : <block_expr>
  | <parser-prod>+
parser-prod ::=
  | <parser-name>? <subrule-prefix>? <subrule> <subrule-suffix>?
  | ~ <ident> <subrule-suffix>?

subrule-prefix ::=
  | &
  | !
subrule-suffix ::=
  | *
  | +
  | ?
subrule-expr ::=
  | parser-rule | subrule-expr
  | parser-rule
subrule ::=
  | { <expr> }
  | ( subrule-expr )
  | <character-set>
  | .
  | <string>
  | <long-ident>
```

The core language

// // About this chapter: // Main author: ? // Paired author:? // // Topics: // - Full syntax // - The database // - Key functions of the standard library //

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.

Lexical conventions

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

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 `/**`.

Documentation

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

{block}[CAUTION] ###### In progress (Soon, a hyperlink to the corresponding chapter) {block}

// TODO TODO The syntax of document comment is described Ill-formed documentation comments do not break the compilation, they only break the documentation.

Basic datatypes

// no need to talk about char I think Opa has 3 basic datatypes: strings, integers and floating point numbers.

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
```

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
```

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"
function 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:

{table} {* 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 |} {table}

Datastructures

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 comma
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 b.c : "po"} //  means {x with b : {x.b 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}}
```

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")
@assert(x == {f1 : 1, f2 : "hello"});
@assert(x.f1 == 1);
@assert({x with f2 : "goodbye"} == (1,"goodbye"));
```

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:

```
list x = {nil}
list x = {hd:3, tl:{hd:4, tl:{hd:5, tl:{nil}}}}
list x = {hd:0, tl:{hd:1, tl:{hd:2, tl:x}}}
```

Identifiers

In Opa, an identifier is a word matched by the following regular expression: `([a-zA-Z_] [a-zA-Z0-9_]* | \\      [ˆ\\\n\\r] \\)except the following keywords:`function,module,with,type,recursive,and,match,if,as,case,default,else,database,parser,_,css,

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: `end`, `external`, `forall`, `import`, `package`, `parser`, `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.

Bindings

At toplevel, you can define an identifier with the following syntax:

```
one = 1
‘hello‘ = "hello"
_z12 = 1+2
```

{block}[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 __. {block}

Of course, local identifiers can be defined too, and they are visible in the following expression:

```
two = {
  one = 1 // semicolon and newline are equivalent
  one + one
}

two = {
  one = 1; one + one // the exact same thing as above
}

two = {
  one = 1 // NOT VALID: syntax error because a local declaration
}         // must be followed by an expression
```

Functions

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:

```
function f(x,y){ // defining function f with the two parameters x and y
  x + y + 1
}

function int f(x,y){ // same as above but explicitly indicate the return type
  x + y + 1
}

two = {
  function f(x){ x + 1 } // functions be defined locally, just like other values
  f(1)
}

// you can write functions in a currified way concisely:
function f(x)(y){ x + y + 1 }
```

{block}[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.

```
function f (){ ... } // WARNING: parsed as an anonymous function which return a value of typ
x = f () // NOT VALID: parse error
```

{block}

Partial applications

From a function with N arguments, we may derive a function with less arguments by *partial application*:

```
function add(x,y){ x+y }
add1 = add(1,_) // which means function add1(y){ add(1,y) }
x = add1(2) // x is 3
```

{block}[CAUTION] Side effects of the arguments are computed at the site and time of partial application, not each time the function is called:

```
add1 = add(loop(), _) // this loops right now
                      // not when calling add1
```

{block}

[[underscore]] All the underscores of a call are regrouped to form the parameters of a unique function in the same order are the corresponding underscores:

```
function max3(x,y,z){ max(x,max(y,z)) }
positive_max = max3(0,_,_) // means function positive_max(x,y){ max(0,x,y) }
```

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:

```
function (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.

```
function{
case 0 : 1
case 1 : 2
case 2 : 3
default : 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).

```
function(e){
  match(e){
  case 0 : 1
  case 1 : 2
  case 2 : 3
  default : error("Wow, that’s outside my capabilities")
  }
}
```

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.

{table} {* leading characters | associativity *} {| | @ | left |} {| || ? | right |} {| & | right |} {| = != > < | left |} {| + - ^ | left |} {| * / | left |} {table}

{block}[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
```

{block}

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:

```
list(int) x = [] // same as s = list(int) []
function list(int) f(x){ [x] } // annotation of the body of the function
                               // same as function f(x){ list(int) [x] }
```

Grouping

Expressions can be grouped with parentheses:

```
x = (1 + 2) * 3
```

Modules

Functionalities are usually regrouped into modules.

```
module List{
  empty = []
  function 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 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:

```
module 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):

```
module M{
  x = y
  y = 1
}
```

This will work just fine, because this is reordered into:

```
module M{
  y = 1
  x = y
}
```

where you have no more recursion.

{block}[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. {block}

Sequencing computations

In Opa the toplevel is executed, and so you can have expressions at the toplevel:

```
println("Executed!")
```

In a block if an expression is not binded and if not the last expression, this expression is computed and the result is discarded.

```
x = {
  println("Dibbs!"); // cleaner than saying _unused_name = println("Dibbs!")
                     // but equivalent (almost, see the warning section)
  println("Aww...");
  1
}
```

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}
@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 = bool {true}
@assert(x.true); // NOT VALID: type error
```

To deconstruct both records and sums, Opa offers *pattern-matching*. The general syntax is:

```
  match(<expr>){
  case <pattern_1> : <expr_1>
  case <pattern_2> : <expr_2>
  ...
  case <pattern_n> : <expr_n>
  default : <expr_default>
  }
```

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 <`expr_default`>. Note the default case (or equivalent `case _`) can be omitted.

The specific case of pattern matching on boolean can be abreviated using a standard `if-then-else` construct:

```
if(1 == 2){
  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
if(1 == 2) println("Who would have known that 1 == 2?")

// or equivalently
match(1 == 2){
case {true} : println("Who would have known that 1 == 2?")
case {false} : println("That's what I thought!")
}
```

{block}[TIP] The same way that `f(x,_)` means (roughly) `function(y){ f(x,y) }`, `_.label` is a shorthand for `function(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]
```

{block}

[[pattern]] ### Patterns

Generally, patterns appear as part of a `match` 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>){case <pattern> :  <expr2> case ...  }` is executed, `<expr>` is evaluated to a value, which is then matched against each pattern in order until a match is found.

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.

{block}[CAUTION] ###### Pattern-matching does not test for equality Consider the following extract:

```
x = 1
y = 2
match(y){
case x : println("Hey, 1=2")
default : 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
match(y){
case z : println("Hey, 1=2")
default : 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. {block}

A few examples:

```
function list_is_empty(l){
  match(l){
  case [] : true
  case [_|_] : false
  }
}

// and without the syntactic sugar for lists
// a list is either {nil} or {hd tl}
function head(l){
  match(list l){
  case {nil} : @fail
  case ~{hd ...} : hd
  }
}
```

{block}[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. {block}

{block}[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. {block}

{block}[WARNING] A pattern cannot contain an expression:

```
function is_zero(x){ // works fine
  match(x){
  case 0 : true
  case _ : false
  }
}

// wrong example
zero = 0
function is_zero(x){
  match(x){
  case zero : true
  case _ : false
  }
}
// does not work because the pattern defines zero
// it does not check that the x is equal to zero
```

{block}

{block}[CAUTION] You cannot put the same variable several times in the same pattern:

```
function on_the_diagonal(position){
  match(position){
  case {x=value y=value} : true
  case _ : false
}
// this is not valid because you are trying to give the name value
// to two values

// this must be written
function on_the_diagonal(position){
  position.x == position.y
}
```

{block}

//Loops //—— // //TODO Rudy - Where are we about 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 .

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{
    case "-"  [0-9]+ : {negative}
    case "+"? [0-9]+ : {positive}
  }
```

A parser is composed of a disjunction of `case <list-of-subrules> (: <semantic-action>)?. When the semantic action is omitted, it defaults to the`text' that was parsed by the left hand side.

A subrule consists of: - an optional binder that names the result of the subrule. It can be: - `x=` to name the result `x` - `~` 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*` - an optional prefix modifier (`!` or `&`) that lookahead for the following subrule in the input - a basic subrule - 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
```

{block}[CAUTION] Putting parentheses around a parser can change the type of the parenthesized parsers:

```
parser{
  case x=.* : ... // x as type list(Unicode.character)
}
parser{
  case 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. {block}

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{
  case "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".

Recursion

By default, toplevel functions and modules are implicitely recursive at toplevel, while local values (including values defined in functions) are not.

```
function f(){ f() } // an infinite loop

x =
  function f(){ f() } // NOT VALID: f is unbound
  void

x =
  recursive function f(){ f() } // now f is visible in its body
  void

function f(){ g() } // mutual recursion works without having
function g(){ f() }// to say 'recursive' anywhere at toplevel

x =
  recursive function f(){ g() } // local mutually recursive functions must
  and function g(){ f() } // be defined together with a 'recursive' '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:

```
module M{
  function f(){ M2.f() }
}
module M2{
  function f(){ m.f() }
}
```

This is valid, because it amounts to:

```
recursive function M_f(){ M2_f() }
and function 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:

```
recursive sess = Session.make(callback)
and function callback(){ /*do something with sess*/ }
```

Please note that the word `recursive` is meant to define *recursive* values, but not meant to define *cyclic* values:

```
recursive x = [x]
```

This definition is invalid, and will be rejected (statically in this case despite the presence of the `recursive` because it is sure to fail at runtime).

Of course, most invalid definitions will be only detected at runtime:

```
recursive x = if(true){ x } else { 0 }
```

Directives

Many special behaviours appear syntactically as directives. - A directive starting with a `@` - Expect the most common directives which are "both" / "server" / "client" / "exposed" / "protected" / "private" / "abstract" 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.

```
if true then void else @fail // @fail appears only in expressions
@expand function ‘=>‘(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. * `@stringifier`:: Takes a typename Overrides the generic stringification for the given type with the function annotated:

```
@stringifier(bool) function to_string(b: bool){ if(b){ "true" } else { "false" } }
```

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
```

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.*
```

{block}[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. {block}

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.

Type expressions

Type expressions are used in type annotations, and in type definitions.

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.

Record types

The syntax for record type works the same as it does in expressions and in patterns:

```
{useless} x = @fail // means {useless:void}
~{a, b} x = @fail // means {a a, b b}, where a and b are typenames
~{list} x = @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)
{a, b, ...} x = {a, b, c} // you can give only a part of the fields in type annotations
```

Tuple types

The type of a tuple actually looks like the tuple:

```
(int,float) (a,b) = (1,3.4)
```

Sum types

Now, record expressions do not have records type (in general), they have sum types, which are simply unions of record types:

```
({true} or {false}) x = {true}
({true} or ...) x = {true} // sum types can be partially specified, just like record types
```

Type names

Types can be given names, and of course you can refer to names in expressions:

```
list(int) x = [1] // the parameters of a type are written just like a function call
bool x = 1 // except that when there is no parameter, you don't write empty parentheses
list x = [1] // and except that you can omit all the parameters of a typename altogether
               // (which means 'please fill up with fresh variables for me')
```

Variables

Variables begin with an apostrophe except _:

```
list('a) x = []
list(_) x = [] // _ is an anonymous variable
```

Function types

Function types are list of types arguments separated by comma then a right arrow precedes type of result:

```
(int, int, int -> int) function max3(x, y, z){
  max(x, max(y, z))
}
```

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} or {false} // naming a type sum
type option('a) = {none} or {'a some} // a parameterized definition of a type sum
type list('a) = {nil} or {'a hd, list('a) tl} // 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
```

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
module Test1{
  Test1.t x = 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 Test1.t = int
module Test1{
  Test1.t x = 1 // will not compile since the module exports
                // Test1.x that has the private type t
}
```

Formal description

This syntax recapitulates the syntactic constructs of the language.

//[|grammar_conventions|] ### 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>
```

The opa language

A source file is a <`program`>, defined as follows:

```
program ::= <declaration>* sep <separator>
declaration ::=
  | <package-declaration>
  | <package-import>
  | <type-definition>
  | <binding>
  | <expr>
```

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>

separator ::=
  | ;
  | \n
```

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 ::=
  | int
  | string
  | float
  | <type-ident> ( <type>* sep , )
  | <record-type>
  | <lambda-type>
  | <sum-type>
  | forall ( <type-var>* sep , ) . <type>
  | <type-var>
  | <tuple-type>
  | ( <type> )

type-ident ::= <long-ident>

record-type ::=
  | ~? { <record-type-field>* sep , <record-type-end>? }
record-type-field ::=
  | <type> <field>
  | ~ <field>
  | <field>
record-type-end ::=
  | , <record-type-var>?
record-type-var ::=
  | ...
  | ' <ident>
```

```
lambda-type ::=
  | <type>* sep , -> <type>

sum-type ::=
  | or? <type> or <type>+ sep or <sum-type-end>?
  | <type> or? <sum-type-var>
sum-type-end ::=
  | or <sum-type-var>?
sum-type-var ::=
  | ...
  | ' <ident>

type-var ::=
  | ' <ident>
  | _

tuple-type ::=
  | ( <type> , )
  | ( <type> , <type>+ sep , ,? )
```

The syntax of the patterns:

```
pattern ::=
  | <literal>
  | <record-pattern>
  | <list-pattern>
  | <tuple-pattern>
  | <pattern> as <ident>
  | <pattern> | <pattern>
  | <type> <pattern>
  | ( <pattern> )
  | _

record-pattern ::=
  | ~? { ...? }
  | ~? { <record-pattern-field>+ sep , ,? ...? }
record-pattern-field ::=
  | ~? <type>? <field>
  | ~? <type>? <field> : <pattern>

list-pattern ::=
 | [ pattern+ sep , | pattern ]
 | [ pattern* sep , ,?]
```

```
tuple-pattern ::=
  | ( <pattern> , )
  | ( <pattern> , <pattern>+ sep , ,? )
```

The syntax of the bindings:

```
binding ::=
  | <binding-directive> recursive <non-rec-binding>+ sep and
  | <binding-directive> <non-rec-binding>

non-rec-binding ::=
  | <ident-binding>
  | <pattern-binding>

ident-binding ::=
  | <type>? <ident> = <expr>
  | <type>? function <type>? <ident> <params>+ { <expr_block> }
  | <type>? module <ident> <params>+ { <non-rec-binding>* sep <separator> }

params ::=
  | ( <pattern>* sep , )

pattern-binding ::=
  | <pattern> = <expr>
```

The syntax of the expressions, except the parsers: // todo database, domaction, xhtml…

```
expr ::=
  | <ident>
  | <expr-or-underscore> <op> <expr-or-underscore>
  | - <expr>
  | <type> <expr>
  | <expr-or-underscore> . field
  | <expr-or-underscore> ( <expr-or-underscore>* sep , )
  | <binding> <separator> <expr>
  | <match>
  | <lambda>
  | <module>
  | <record>
  | { <expr_block> }
  | ( expr )
  | <tuple>
  | <list>
  | <literal>
  | <directive>
  | <sysbinding>
  | <parser>

expr_block ::=
  | <expr>
  | <expr_or_binding>+ sep <separator> <separator> <expr>
expr_or_binding ::=
  | <expr>
  | <binding>

match ::=
  | match ( <expr> ){ <match_case>* <match_default> }
  | if <expr> then <expr> else <expr>
  | if <expr> then <expr>
match_case ::=
  | case <pattern> : <expr_block>
  | default : <expr_block>

lambda ::=
  | function <type>? <space> <params>+ { <expr_block> }
  | function{ <match_case>* <match_default> }

record ::=
  | ~? { <record-field>* sep ,? }
  | ~? { <expr> with <record-field>+ sep ;? }
record-field ::=
  | ~? <type>? <field>
  | <type>? <field> : <expr>

tuple ::=
  | ( <expr> , )
  | ( <expr> , <expr>+ sep , ,? )

list ::=
  | [ <expr>+ sep , | <expr> ]
  | [ <expr>* sep , ,? ]

module ::=
  | module{ <non-rec-binding>+ sep <separator> }

expr-or-underscore ::=
  | <expr>
  | _
```

The syntax of the parsers:

```
parser ::=
  | parser{ <parser-case>+ }
parser-case ::=
 | case <parser-rule>
parser-rule ::=
  | <parser-prod>+ : <block_expr>
  | <parser-prod>+
parser-prod ::=
  | <parser-name>? <subrule-prefix>? <subrule> <subrule-suffix>?
  | ~ <ident> <subrule-suffix>?

subrule-prefix ::=
  | &
  | !
subrule-suffix ::=
  | *
  | +
  | ?
subrule-expr ::=
  | parser-rule | subrule-expr
  | parser-rule
subrule ::=
  | { <expr> }
  | ( subrule-expr )
  | <character-set>
  | .
  | <string>
  | <long-ident>
```

Developing for the web

// // About this chapter: // Main author: Valentin Gatien-Baron // Paired author: ? // // Topics: // - XHTML/XML syntax, types [xmlns], [xhtml] // - Slicer syntax and semantics // - xml_parser // - CSS syntax // - including resources // - more?

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.

Syntax extensions

//Since Opa is meant for web development, there are some syntactic constructs that are meant for this purpose, //and so have not been introduced in the core language.

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.

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:

```
(xhtml)
  { namespace: Xhtml.ns_uri
  , tag: "div"
  , args: []
  , specific_attributes: {
      some: {
        class: ["something"],
        style: [],
        events: [],
        events_options: [],
        href: {none}
      }
    }
  , content: [{ text: "Hey" }]
  }
```

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} </>
```

{block}[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
```

{block}

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)`.

//do we care?.events // don't get what it does

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>
)
```

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 {
    case <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.

//ATTENTION avec l'histoire des whitespaces //When some xml is given to a parser, it first flattens the xml (in case it is a fragment) and it discard text nodes //that contain only whitespace. //In the previous example, we get the list `[<x>12</>, <y>13</>]`. + //The result is then given to the first subrule `<x>parser x=Rule.integer</>`. + //This subrule binds `x` to `12` and return the remaining nodes `[<y>13</>]`. + //These remaining nodes are then given to the second subrule `<y>parser y=Rule.integer</>`. + //This subrule binds `y` to `13` and return the remaining nodes `[]`. + //The action corresponding to the pattern is then executed. +

//We detail here the meaning 'a list of xml matches a pattern.' 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 and the result of 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 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. If no node is matched, `{none}` is returned.
- An xml matches the subrule `<subrule>{min,max}` when the xml matches 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 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 . In that case, the value of the matching is bound to `<ident>` instead of being bound to the name of the attribute.

{block}[IMPORTANT] All whitespace only text nodes are discarded during parsing.

```
xml = @xml(<>"  "</>)
p = xml_parser {
      case parser .*: {}
    }
```

`p` would fail at parsing `xml`, because it behaves exactly as if `xml` were defined as

```
xml = @xml(<></>)
```

{block}

// An xml matches a list of subrules if // Any list of xml matches the empty list pattern. (<></>) // A list of xml matches a non empty list pattern if the list of xml matches the head // and if the remaining nodes matches the tail pattern.

// 2. A list of xml match a subrule:

// A subrule can be: // - _ : matches exactly one text node or a tag node // If the result of this pattern is named, it will have type +xhtml+. // - content </> // this pattern matches the first node of the list if the first node // if a tag node with the same tag, matching attributes and matching content // The content is considered to match even where there are remaining nodes in the values // that are not matched by the pattern // The attributes also match even when there more attributes in the values that in pattern. // For instance matches any tag node with the name div, with any quantity of children // and with any attribute. // If the result of this pattern is named, it will have type `xhtml`. // - an expression between curly braces is simply another xml parser that will be called on the // current list of nodes (that may read as many nodes as it wishes) // If the result of this pattern is named, it will have the return type of parser. // - parser matches when the first node of the list is a text node // and when the parser successfully parser the content of text node. // If the result of this pattern is named, it will have the return type of the parser. // - If the subrule is omitted, it defaults to _

// An attribute pattern can be: // - attr=_ // Matches when the tag node has a the attribute attr, without conditions on its value. // - attr // Matches when the tag node has a the attribute attr, and binds the value to the name `attr` // - attr=({e} as name) // e should be a `Parser.general('a)` this constructs binds name to the value returned by the parser // (of type `'a`) // - attr=(as name) // shorthand for attr=({Parser.of_string(}} as name) // - attr={e} // shorthand for attr=({e} as attr) // - attr= // shorthand for attr=(as attr)

// You can put modifiers on the subrules: // - subrule* means to match (greedily) as many consecutive `subrule` as possible // If the result of this pattern is named, its type will a be a list of the type returned by subrule. // - subrule`means to match (greedily) as many consecutive`subrule`as possible // and the result should be non empty // If the result of this pattern is named, its type will a be a list of the type returned by subrule // (and the list is guaranteed to be non empty).  // - subrule{min,max} means to match (greedily) up to max consecutive`subrule', // but to fail is less that min were parsed // If the result of this pattern is named, its type will a be a list of the type returned by subrule // (and the list is guaranteed to be at least min long and at most max long). // - subrule{number} is a shorthand for subrule{number,number}

Ip address

You can write constant ip addresses with the usual syntax:

```
127.0.0.1
```

This expression has type `IPv4.ip`.

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.
- `@xml` :: Takes an xml literal. The default namespace in the literal in the empty uri, not the xhtml uri. //@static_content:: //@static_content_directory:: //@static_include_directory:: //@static_source_content:: //@static_binary_content:: // TODO! //* `@static_resource` :: //* `@static_resource_directory` ::

The set of binding directives is also extended: //* `both_implem` ? // TODO: explain! [[web_binding_directives]] * `public` :: * `private` :: * `both` :: * `client` :: * `server` :: * `exposed` :: * `protected` :: * `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. * `comparator` :: * `stringifier` :: * `xmlizer` :: Takes a typename. Overrides the generic transformtion to xml for the given type with the function annotated.

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; } }>
```

//css = css css_declaration -> lifted to a list of length one // | expr -> of type list(css_declaration) //css css_map //css { } //style=‘’ //% // but not available since % is already an expression // (type Css.length) // // // but not available since it is parsed as an identifier //#code couleur // ok car # est pas parsable autrement //rgb(..,..,..) // pas ok car c'est un appel de fonction d'abord //url("url") // not available same reason //url('url') // available because it can not be parsed as a function call //Css.prop_value_expr_opa

Defining servers

The way to define a server in Opa is by mean of the function `Server.start`, where the first argument is the server configuration and the second one is a handler for the URLs (with many possible variants).

```
Server.start(Server.http,
  { title: "Hello"
  , page: function() { <h1>Hello World</h1> }
  }
)
```

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.

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.

```
Dom.transform([#show_messages = <>{failure}</>])
```

If the list of actions to perform contains only one single element then the `Dom.transform` application can be ommitted and the above can be replaced with simple

```
#show_messages = <>{failure}</>
```

Client-server distribution

// .About this section // *********************** // This section details the distribution between client and server, including:

// - the rules for client/server slicing; // - the semantics of serialization client <-> server serialization; // - the semantics of (dis)connexion; // - the semantics of garbage-collection; // - caching; // - best practices. // *********************** This section details the distribution between client and server.

// // Main editor for this section: Valentin Gatien-Baron //

// // If an item spans several sections, please provide // hyperlinks, e.g. type definitions have both a syntax // and a more complete definition on the corresponding // section //

// // If an item is considered experimental and may or may // not survive to future versions, please label it using // an Admonition block with style [CAUTION] //

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 (than a per-function) granularity.

The slicer can be told where a declaration should end up with the slicing annotations put before the `function` keyword:

- `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. // and reading of the global state (like getting the current time). 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(. . .)

// FIXME: both_implem does not work, @both_implem does – parser should be fixed.

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 directive is:

- `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: in `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`:

// FIXME: sliced_expr does not seem to work with the new syntax.

```
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).

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. //Integers, floats and strings and records will be copied.

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`; see here). 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.

//Flow of control and remote call //^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ // il faut présenter le publish_async // il faut aussi dire que les appels client->server sont bloquants // sauf ceux qui sont publish async // même comme ça, on a un seul flot sur le server apparemment donc // l’éxecution reste sequentielle sur le server

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: [[syntax_keyword_static]]

- `@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: [[syntax_keyword_static_directory]]

- `@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(key)`. 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.

Examples

The two typical scenarios are embedding one resource:

```
handler = parser {
  case "/favicon.ico": @static_resource("img/favicon.ico")
  case "/favicon.gif": @static_resource("img/favicon.gif")
}
```

and embedding many resources:

```
resources = @static_resource_directory("resources")
urls      = parser {
              case "/": start
              case resource={Server.resource_map(resources)}: resource
            }
Server.start(Server.http, {custom: 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.

See also

Resources embedded with these directives support runtime modification for debugging purposes. For more details, see the related section.

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 {
              case "/": start
              case resource={Server.permanent_resource_map(resources)}: resource
            }
Server.start(Server.http, {custom: 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 .

//### Index of keywords and directives //- //- //- //-

Developing for the web

// // About this chapter: // Main author: Valentin Gatien-Baron // Paired author: ? // // Topics: // - XHTML/XML syntax, types [xmlns], [xhtml] // - Slicer syntax and semantics // - xml_parser // - CSS syntax // - including resources // - more?

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.

Syntax extensions

//Since Opa is meant for web development, there are some syntactic constructs that are meant for this purpose, //and so have not been introduced in the core language.

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.

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:

```
(xhtml)
  { namespace: Xhtml.ns_uri
  , tag: "div"
  , args: []
  , specific_attributes: {
      some: {
        class: ["something"],
        style: [],
        events: [],
        events_options: [],
        href: {none}
      }
    }
  , content: [{ text: "Hey" }]
  }
```

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} </>
```

{block}[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
```

{block}

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)`.

//do we care?.events // don't get what it does

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>
)
```

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 {
    case <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.

//ATTENTION avec l'histoire des whitespaces //When some xml is given to a parser, it first flattens the xml (in case it is a fragment) and it discard text nodes //that contain only whitespace. //In the previous example, we get the list `[<x>12</>, <y>13</>]`. + //The result is then given to the first subrule `<x>parser x=Rule.integer</>`. + //This subrule binds `x` to `12` and return the remaining nodes `[<y>13</>]`. + //These remaining nodes are then given to the second subrule `<y>parser y=Rule.integer</>`. + //This subrule binds `y` to `13` and return the remaining nodes `[]`. + //The action corresponding to the pattern is then executed. +

//We detail here the meaning 'a list of xml matches a pattern.' 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 and the result of 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 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. If no node is matched, `{none}` is returned.
- An xml matches the subrule `<subrule>{min,max}` when the xml matches 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 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 . In that case, the value of the matching is bound to <`ident`> instead of being bound to the name of the attribute.

{block}[IMPORTANT] All whitespace only text nodes are discarded during parsing.

```
xml = @xml(<>"  "</>)
p = xml_parser {
      case parser .*: {}
    }
```

`p` would fail at parsing `xml`, because it behaves exactly as if `xml` were defined as

```
xml = @xml(<></>)
```

{block}

// An xml matches a list of subrules if // Any list of xml matches the empty list pattern. (<></>) // A list of xml matches a non empty list pattern if the list of xml matches the head // and if the remaining nodes matches the tail pattern.

// 2. A list of xml match a subrule:

// A subrule can be: // - _ : matches exactly one text node or a tag node // If the result of this pattern is named, it will have type +xhtml+. // - content </> // this pattern matches the first node of the list if the first node // if a tag node with the same tag, matching attributes and matching content // The content is considered to match even where there are remaining nodes in the values // that are not matched by the pattern // The attributes also match even when there more attributes in the values that in pattern. // For instance matches any tag node with the name div, with any quantity of children // and with any attribute. // If the result of this pattern is named, it will have type `xhtml`. // - an expression between curly braces is simply another xml parser that will be called on the // current list of nodes (that may read as many nodes as it wishes) // If the result of this pattern is named, it will have the return type of parser. // - parser matches when the first node of the list is a text node // and when the parser successfully parser the content of text node. // If the result of this pattern is named, it will have the return type of the parser. // - If the subrule is omitted, it defaults to _

// An attribute pattern can be: // - attr=_ // Matches when the tag node has a the attribute attr, without conditions on its value. // - attr // Matches when the tag node has a the attribute attr, and binds the value to the name `attr` // - attr=({e} as name) // e should be a `Parser.general('a)` this constructs binds name to the value returned by the parser // (of type `'a`) // - attr=(as name) // shorthand for attr=({Parser.of_string(}} as name) // - attr={e} // shorthand for attr=({e} as attr) // - attr= // shorthand for attr=(as attr)

// You can put modifiers on the subrules: // - subrule* means to match (greedily) as many consecutive `subrule` as possible // If the result of this pattern is named, its type will a be a list of the type returned by subrule. // - subrule`means to match (greedily) as many consecutive`subrule`as possible // and the result should be non empty // If the result of this pattern is named, its type will a be a list of the type returned by subrule // (and the list is guaranteed to be non empty). // - subrule{min,max} means to match (greedily) up to max consecutive`subrule', // but to fail is less that min were parsed // If the result of this pattern is named, its type will a be a list of the type returned by subrule // (and the list is guaranteed to be at least min long and at most max long). // - subrule{number} is a shorthand for subrule{number,number}

Ip address

You can write constant ip addresses with the usual syntax:

```
127.0.0.1
```

This expression has type `IPv4.ip`.

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.
- `@xml` :: Takes an xml literal. The default namespace in the literal in the empty uri, not the xhtml uri. //@static_content:: //@static_content_directory:: //@static_include_directory:: //@static_source_content:: //@static_binary_content:: // TODO! //* `@static_resource` :: //* `@static_resource_directory` ::

The set of binding directives is also extended: //* `both_implem` ? // TODO: explain! [[web_binding_directives]] * `public` :: * `private` :: * `both` :: * `client` :: * `server` :: * `exposed` :: * `protected` :: * `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. * `comparator` :: * `stringifier` :: * `xmlizer` :: Takes a typename. Overrides the generic transformtion to xml for the given type with the function annotated.

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; } }>
```

//css = css css_declaration -> lifted to a list of length one // | expr -> of type list(css_declaration) //css css_map //css { } //style=‘’ //% // but not available since % is already an expression // (type Css.length) // // // but not available since it is parsed as an identifier //#code couleur // ok car # est pas parsable autrement //rgb(..,..,..) // pas ok car c'est un appel de fonction d'abord //url("url") // not available same reason //url('url') // available because it can not be parsed as a function call //Css.prop_value_expr_opa

Defining servers

The way to define a server in Opa is by mean of the function `Server.start`, where the first argument is the server configuration and the second one is a handler for the URLs (with many possible variants).

```
Server.start(Server.http,
  { title: "Hello"
  , page: function() { <h1>Hello World</h1> }
  }
)
```

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.

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.

```
Dom.transform([#show_messages = <>{failure}</>])
```

If the list of actions to perform contains only one single element then the `Dom.transform` application can be ommitted and the above can be replaced with simple

```
#show_messages = <>{failure}</>
```

Client-server distribution

// .About this section // ********************** // This section details the distribution between client and server, including:

// - the rules for client/server slicing; // - the semantics of serialization client <-> server serialization; // - the semantics of (dis)connexion; // - the semantics of garbage-collection; // - caching; // - best practices. // ********************** This section details the distribution between client and server.

// // Main editor for this section: Valentin Gatien-Baron //

// // If an item spans several sections, please provide // hyperlinks, e.g. type definitions have both a syntax // and a more complete definition on the corresponding // section //

// // If an item is considered experimental and may or may // not survive to future versions, please label it using // an Admonition block with style [CAUTION] //

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 (than a per-function) granularity.

The slicer can be told where a declaration should end up with the slicing annotations put before the `function` keyword:

- `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. // and reading of the global state (like getting the current time). 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(. . .)

// FIXME: both_implem does not work, @both_implem does – parser should be fixed.

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 directive is:

- `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: in `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`:

// FIXME: sliced_expr does not seem to work with the new syntax.

```
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).

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. //Integers, floats and strings and records will be copied.

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`; see here). 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.

//Flow of control and remote call //^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ // il faut présenter le publish_async // il faut aussi dire que les appels client->server sont bloquants // sauf ceux qui sont publish async // même comme ça, on a un seul flot sur le server apparemment donc // l'éxecution reste sequentielle sur le server

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: [[syntax_keyword_static]]

- `@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: [[syntax_keyword_static_directory]]

- `@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(key)`. 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.

Examples

The two typical scenarios are embedding one resource:

```
handler = parser {
  case "/favicon.ico": @static_resource("img/favicon.ico")
  case "/favicon.gif": @static_resource("img/favicon.gif")
}
```

and embedding many resources:

```
resources = @static_resource_directory("resources")
urls      = parser {
              case "/": start
              case resource={Server.resource_map(resources)}: resource
            }
Server.start(Server.http, {custom: 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.

See also

Resources embedded with these directives support runtime modification for debugging purposes. For more details, see the related section.

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 {
              case "/": start
              case resource={Server.permanent_resource_map(resources)}: resource
            }
Server.start(Server.http, {custom: 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 .

//### Index of keywords and directives //- //- //- //-

The database

Database operations are fully integrated in the Opa language, which increases code re-usability, simplifies code coverage tests, and permits both automated optimizations and automated safety, security and sanity checks.

In this chapter, we describe the constructions of Opa dedicated to storage of data and manipulation of stored data.

Declaring a database

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 and remove data.

A database is named and consists of a set of declared paths. Moreover Opa handles several database backends; currently: - *MongoDb*: The popular MongoDB database (http://www.mongodb.org/); - *Db3*: The native Opa database which allows for rapid prototyping of applications.

The following piece of code declares a database named `my_db` and defines : - 3 paths containing basic values (/i, /f, /s) - 1 path containing a record (/r) - 1 path containing a list of strings - 1 path to a map from integers (`intmap`) to values of type `stored`. - 1 path to a set of `stored` records, where the primary key of the declared set is `x`. - 1 path to a set of more complex records where the primary key of the set is `id`. This most complex records contain embedded maps and lists.

```
type stored = {int x, int y, string v, list(string) lst}

// where _db\_options_ allows to select the database backend, see below
database my_db [db_options] {
  int    /basic/i
  float  /basic/f
  string /basic/s
  stored /r
  intmap(stored) /imap
  stored /set[{x}]
  {int id, stored s, intmap(stored) imap, stringmap(stored) smap} /complex[{id}]
}
```

Choosing database backend

Database backend depends on presence and value of `db_options`: - `@mongo`: selects MongoDb backend - `@db3` value selects Db3 backend - If the db_options is omitted, the selected backend depends on the compiler '–database' command line option, with `Db3` being the default (soon to be changed to MongoDB).

Initializing database

Opa applications can use several databases with several different backends. For each database a command line family is generated. When running your applications you can specify, via command line arguments, options for each declared database.

Both backends have common command lines options : - `--db-local:dbname [/path/to/db]`: Use a local database, stored at the specified location in the file-system. - `--db-remote:dbname host(s)`: Use a remote database accessible at a given remote location.

Note that when you use for the first time the local database options with MongoDb, the Opa application will download, install, and launch a single instance of MongoDb database.

Note also that you can define database authentication parameters for each database with the syntax `--db-remote:dbname username:password@hostname:port`.

Database reads/writes

One can read from the `/my_db/basic/i` path with:

```
x = /my_db/basic/i
```

and write the contents of said path:

```
/my_db/basic/i <- x
```

Note that the database is structured. Therefore, if you access path `/my_db/basic` you will obtain a value of type { `int i, float f, string s` }.

Conversely, you could have defined the path `/my_db/basic` as:

```
...
{ int i, float f, string s } /basic
...
```

and then still access paths `/my_db/basic/i`, `/my_db/basic/f` or `/my_db/basic/s` directly.

Default values

Each path has a *default value*. Whenever you attempt to read a value that does not exist (was never initialized or was removed), the default value is returned.

While Opa can generally define a default value automatically, it is often a good idea to define an application-specific default value:

```
...
string /basic/s = "default string"
...
```

If you do not define a default value, Opa uses the following strategy:

- the default value for an integer (`int`) is `0`;
- the default value for a floating-point number (`float`) 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.)

Sub-paths

Sub-paths are paths relative to some location in the database. If you think about paths as akin to absolute paths in the file-system, then sub-paths are a bit like relative paths.

There are several ways to build sub-paths, that can be further composed by by separating them with dots (`.`, as opposed to `/` used for paths).

- A *record field* (`<ident>`) accessing a record field with the given name,
- An *indexed_expression* (`[<expr>]`) accessing elements of a map with constaints given by `<expr>` and
- A *hole expression* (`[_]`) acessing elements of a list.

We'll talk more about sub-path in the following chapters.

Querying

Opa 0.9.0 (S4) introduces database sets and a powerful way to access a subset of database collections.

A query is composed from the following operators: - `== expr`: equals expr - `!= expr`: not equals expr - `< expr`: lesser than expr - `<= expr`: lesser than or equals expr - `> expr`: greater than expr - `>= expr`: greater than or equals expr - `in expr`: "belongs to" `expr`, where `expr` is a list - `q1 or q2`: satisfy query q1 or q2 - `q1 and q2`: satisfy queries q1 and q2 - `not q`: does not satisfy q - `{f1 q1, f2 q2, ...}`: the database field `f1` satisfies `q1`, field `f2` satisfies `q2` etc.

Furthermore you can specify some querying options: - `skip n`: where `expr` should be an expression of type int, skip the first `n` results - `limit n`: where `expr` should be an expression of type int, returns a maximum of `n` results - `order fld (, fld)+`: where fld specify an order. `fld` can be a single identifier or a list of identifiers specifying the fields on which the ordering should be based. Identifiers can optionally be prefixed with `+` or `-` to specify, respectively, ascending or descending order. Finally it's possible to choose the order dynamically with `<ident>=<expr>` where `<expr>` should be of type `{up} or {down}`.

Querying database sets

```
k = {x : 9}
stored x = /dbname/set[k] // k should be type of set keys, returns a unique value
stored x = /dbname/set[x == 10] // Returns a unique value because {x} is the primary key of

dbset(stored, _) x = /dbname/set[y == 10] // Returns a database set because y is not a prima
dbset(stored, _) x = /dbname/set[x > 10, y > 0, v in ["foo", "bar"]]

dbset(stored, _) x = /dbname/set[x > 10, y > 0, v in ["foo", "bar"]; skip 10; limit 15; orde

it = DbSet.iterator(x); // then use Iter module.
```

Querying database maps

```
// Access to a unique value (like in Opa <0.9.0 (S3))
stored x = /dbname/imap[9]

// Access to a submap where keys are smaller than 9 and greater than 4
intmap(stored) submap = /dbname/imap[< 9 and > 4]
```

Sub-queries

We can also use sub-paths to perform sub-queries on elements at a given path.

With indexed-expressions you can query on values inside maps.

```
// Querying using expression fields.
// This query returns all elements in the database set at path
// `/dbname/complex` where
// - The field `x` of the `stored` record on field `s` of database set
//   elements is smaller that 100
// - and the intmap `imap` contains an elements associated to the key `10`
//   where the field `v` is equal to `"value"`
// - and the stringmap `smap` contains a bindings with the key `"key"`
dbset(_, _) x = /dbname/complex[s.x < 100 and imap[10].v == "value" and smap["key"] exists]
```

With hole expressions you can query elements within a list.

```
// Querying using hole expressions.
// This query returns all elements for which the stringmap `smap` contains a
// value associated to the `"key"` and the list on the field `lst` contains a
// value equal to "value".
dbset(_, _) x = /dbname/complex[smap["key"].lst[_] == "value"]
```

Updating

Previously we saw how to overwrite a path, like that :

```
/path/to/data <- x
```

which assigns to path `/path/to/data` the value of `x`. This used to be the only way to write to a path, but since Opa 0.9.0 (S4) there are more ways to update them.

Int updating

```
// Overwrite
/my_db/basic/i <- 42

// Increment
/my_db/basic/i++
/dbname/i += 10

// Decrement
/my_db/basic/i--
/my_db/basic/i -= 10

// Sure we can go across records
/my_db/r/x++
```

Record updating

```
// Overwrite an entire record
x = {x: 1, y: 2, v: "Hello, world", lst: []}
/my_db/r <- x

// Update a subset of record fields
/my_db/r <- {x++, y--}

/my_db/r <- {x++, v : "I just want to update z and incr x"}
```

List updating

```
// Overwrite an entire list
/my_db/l <- ["Hello", ",", "world", "."]

// Removes first element of a list
/my_db/l pop

// Removes last element of a list
/my_db/l shift
```

```
// Append an element
/my_db/l <+ "element"

// Append several elements
/my_db/l <++ ["I", "love", "Opa"]
```

Set/Map updating

The values stored inside database sets and maps can be updated as above. The difference is how we access the elements (more details in the querying section). Furthermore updates can operates on several paths.

```
//Increment the field y of the record stored at position 9 of imap
/dbname/imap[9] <- {y++}

//Increment fields y of records stored with key smaller than 9
/dbname/imap[< 9] <- {y++}

//Increment the field y of record where the field x is equals 9
/dbname/set[x == 9] <- {y++}

//Increment the field y of all records where the field x is smaller than 9
/dbname/set[x < 9] <- {y++}
```

Updating sub-paths

With sub-paths it is possible to update some values deeper in the database structure, relative to the point of update.

For instance, with indexed-expressions it's possible to write:

```
// Updating using expression fields.
// Update elements which match the query ‘...‘ by:
// - Incrementing field ‘x‘ of the ‘stored‘ record on field ‘s‘ of the database set
// - and add 2 to the field ‘x‘ and substract 2 from the field ‘y‘ of the
//   element associated to the ‘"key"‘ in the stringmap ‘smap‘
/dbname/complex[...] <- {s.x++, smap["key"].{x += 2, y -= 2}}
```

With hole expressions it is possible to update *the first* element of a list matched by the corresponding hole in the query.

```
// Updating using hole fields.
// Write "another" into the first element of the list ‘lst‘ in the map ‘smap‘
// at key "key".
/dbname/complex[smap["key"].lst[_] == "value"] <- {smap["key"].lst[_] : "another"}
```

Projection

We saw how we can query database sets and maps to obtain a sub-set of their values. But often we do not need all the information stored there but only a selected few fields and fetching complete information from the database would be inefficient.

We already saw how we can access a subpath of a given path:

```
// Access to /my_db/r
stored x = /my_db/r
// Access to the /x sub-path of /my_db/r
int x = /my_db/r/x
```

If we want to access to both subpath /x and /y of path /my_db/r we can write something like that:

```
{int x, int y} xy = {x : /my_db/r/x, y : /my_db/r/y}
```

But there is a more concise syntax to achieve the same:

```
// This path access means 'select fields x and y in path /my_db/r'
{int x, int y} x = /my_db/r.{x, y}
```

Similarly, you can use these kinds of projections on database sets and maps. Some examples follow:

```
// Access of /v sub-path of a set access with primary key
string result = /dbname/set[x == 10]/v

// Access of both sub-path /x and /v of a set access with primary key
{int x, string v} result = /dbname/set[x == 10].{x, v}

// Access of /v sub-paths of a set access
dbset(string, _) results = /dbname/set[x > 10, y > 0, v in ["foo", "bar"]]/v

// Access of two sub-paths /x and /v of a set access
dbset({int x, string v}, _) results = /dbname/set[x > 10, y > 0, v in ["foo", "bar"]].{x, v
```

You can also use sub-paths to project selected data.

```
dbset({{int x, int y} s, intmap(stored) imap}) results = /dbname/complex[...].{s.{x, y}, ima

dbset({{int x, int y} s, stored imap}) results = /dbname/complex[...].{s.{x, y}, imap["key"]
```

Manipulating paths

Most operations deal with data, as described above. We will now describe operations that deal with meta-data, that is the paths themselves.

A path written with notation `!/path/to/data` is called a *value path*. Value paths represent a snapshot of the data stored at that path. Should you write a new value at this path at a later stage, this will not affect the data or meta-data of such snapshots.

```
// Getting a value path. i.e. a read only path
!/path/to/data
```

Should you need to make reference to the *latest version* of data and meta-data, you will need to use a *reference path*, as obtained with the following notation:

```
// Getting a reference path. i.e. a r/w path
@/path/to/data
```

A path value is specific to its own database backend, indeed the type of a path contains 2 type variables; the first one for the type of data stored in the path and the second one for specifying the database backends used for this path.

For instance if `my_db` is declared as a MongoDb database, here is type of its paths :

```
Db.val_path(string, DbMongo.engine) !/my_db/s
Db.ref_path(string, DbMongo.engine) @/my_db/s
```

Generic functions to manipulate such values are available in the `Db` module (imported by default). There are also more functions specific to different backends: - `Db3` module from package `stdlib.database.db3` for Db3 backend features, such as path history. - `DbMongo` from package `stdlib.database.mongo` for MongoDb backend features.

Note, that the same mechanism is used for `dbset('data, 'engine)`

Db3 backend specifics

Restrictions :

- Most of querying/updating are not yet implemented

Db3 Transactions

Whenever database paths can be accessed simultaneously by many users, consistency needs to be ensured. For this purpose, Opa offers a mechanism of *transactions*. Apart from consistency it is also more efficient to explicitly encapsulate a sequence of database operations occurring in a row within a single transaction.

In their simplest and most common form, transactions take the form of function `Db.transaction`:

```
function atomic() {
    //...any operations
    void
}

result = Db.transaction(atomic)
match (result) {
 case {none}: //a conflict or another database error prevented the commit
 case ~{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 does not 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()

tr.in(atomic)

// some other treatments

match (tr.commit()) {
 case {success}: // ...
 case {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 from 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.

// Note: conflict resolution // At the time being, a transaction will conflict whenever some data that it writes // has been changed in the meantime. Other conflict policies are planned and, in the future, // it will be possible to select them on specific database paths (eg. conflict if // the transaction *read* some data that has been changed at the time of commit, // solve conflicts on a counter by adding the increments, etc.)

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.

Database Schema

Db3 verify that the database stored to the disk is compatible with the application. If the database schema has changed, Opa will offer the possibility to perform an automated database migration.

MongoDb backend specifics

Currently, the MongoDb driver does not verify if the compiled application is compatible with a database schema (the way the Db3 does). That means that after changing data used in the application the developer herself needs to take care of migrating the data.

The database

Database operations are fully integrated in the Opa language, which increases code re-usability, simplifies code coverage tests, and permits both automated optimizations and automated safety, security and sanity checks.

In this chapter, we describe the constructions of Opa dedicated to storage of data and manipulation of stored data.

Declaring a database

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 and remove data.

A database is named and consists of a set of declared paths. Moreover Opa handles several database backends; currently: - *MongoDb*: The popular MongoDB database (http://www.mongodb.org/); - *Db3*: The native Opa database which allows for rapid prototyping of applications.

The following piece of code declares a database named `my_db` and defines : - 3 paths containing basic values (/i, /f, /s) - 1 path containing a record (/r) - 1 path containing a list of strings - 1 path to a map from integers (`intmap`) to values of type `stored`. - 1 path to a set of `stored` records, where the primary key of the declared set is `x`. - 1 path to a set of more complex records where the primary key of the set is `id`. This most complex records contain embedded maps and lists.

```
type stored = {int x, int y, string v, list(string) lst}

// where _db\_options_ allows to select the database backend, see below
database my_db [db_options] {
  int    /basic/i
  float  /basic/f
  string /basic/s
  stored /r
  intmap(stored) /imap
  stored /set[{x}]
  {int id, stored s, intmap(stored) imap, stringmap(stored) smap} /complex[{id}]
}
```

Choosing database backend

Database backend depends on presence and value of `db_options`: - `@mongo`: selects MongoDb backend - `@db3` value selects Db3 backend - If the db_options is omitted, the selected backend depends on the compiler '–database' command line option, with `Db3` being the default (soon to be changed to MongoDB).

Initializing database

Opa applications can use several databases with several different backends. For each database a command line family is generated. When running your applications you can specify, via command line arguments, options for each declared database.

Both backends have common command lines options : - `--db-local:dbname [/path/to/db]`: Use a local database, stored at the specified location in the file-system. - `--db-remote:dbname host(s)`: Use a remote database accessible at a given remote location.

Note that when you use for the first time the local database options with MongoDb, the Opa application will download, install, and launch a single instance of MongoDb database.

Note also that you can define database authentication parameters for each database with the syntax `--db-remote:dbname username:password@hostname:port`.

Database reads/writes

One can read from the `/my_db/basic/i` path with:

```
x = /my_db/basic/i
```

and write the contents of said path:

```
/my_db/basic/i <- x
```

Note that the database is structured. Therefore, if you access path `/my_db/basic` you will obtain a value of type `{ int i, float f, string s }`.

Conversely, you could have defined the path `/my_db/basic` as:

```
...
{ int i, float f, string s } /basic
...
```

and then still access paths `/my_db/basic/i`, `/my_db/basic/f` or `/my_db/basic/s` directly.

Default values

Each path has a *default value*. Whenever you attempt to read a value that does not exist (was never initialized or was removed), the default value is returned.

While Opa can generally define a default value automatically, it is often a good idea to define an application-specific default value:

```
...
string /basic/s = "default string"
...
```

If you do not define a default value, Opa uses the following strategy:

- the default value for an integer (`int`) is `0`;
- the default value for a floating-point number (`float`) 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.)

Sub-paths

Sub-paths are paths relative to some location in the database. If you think about paths as akin to absolute paths in the file-system, then sub-paths are a bit like relative paths.

There are several ways to build sub-paths, that can be further composed by by separating them with dots (`.`, as opposed to `/` used for paths).

- A *record field* (<`ident`>) accessing a record field with the given name,
- An *indexed_expression* (`[`<`expr`>`]`) accessing elements of a map with constaints given by <`expr`> and
- A *hole expression* (`[_]`) acessing elements of a list.

We'll talk more about sub-path in the following chapters.

Querying

Opa 0.9.0 (S4) introduces database sets and a powerful way to access a subset of database collections.

A query is composed from the following operators: - `== expr`: equals expr - `!= expr`: not equals expr - < `expr`: lesser than expr - <= `expr`: lesser than or equals expr - > `expr`: greater than expr - >= `expr`: greater than or equals expr - `in expr`: "belongs to" `expr`, where `expr` is a list - `q1 or q2`: satisfy query q1 or q2 - `q1 and q2`: satisfy queries q1 and q2 - `not q`: does not satisfy q - {`f1 q1, f2 q2, ...`}: the database field `f1` satisfies `q1`, field `f2` satisfies `q2` etc.

Furthermore you can specify some querying options: `- skip n`: where `expr` should be an expression of type int, skip the first `n` results - `limit n`: where `expr` should be an expression of type int, returns a maximum of `n` results - `order fld (, fld)+`: where fld specify an order. `fld` can be a single identifier or a list of identifiers specifying the fields on which the ordering should be based. Identifiers can optionally be prefixed with `+` or `-` to specify, respectively, ascending or descending order. Finally it's possible to choose the order dynamically with <`ident`>=<`expr`> where <`expr`> should be of type {`up`} `or` {`down`}.

Querying database sets

```
k = {x : 9}
stored x = /dbname/set[k] // k should be type of set keys, returns a unique value
stored x = /dbname/set[x == 10] // Returns a unique value because {x} is the primary key of

dbset(stored, _) x = /dbname/set[y == 10] // Returns a database set because y is not a prima

dbset(stored, _) x = /dbname/set[x > 10, y > 0, v in ["foo", "bar"]]

dbset(stored, _) x = /dbname/set[x > 10, y > 0, v in ["foo", "bar"]; skip 10; limit 15; orde

it = DbSet.iterator(x); // then use Iter module.
```

Querying database maps

```
// Access to a unique value (like in Opa <0.9.0 (S3))
stored x = /dbname/imap[9]

// Access to a submap where keys are smaller than 9 and greater than 4
intmap(stored) submap = /dbname/imap[< 9 and > 4]
```

Sub-queries

We can also use sub-paths to perform sub-queries on elements at a given path.

With indexed-expressions you can query on values inside maps.

```
// Querying using expression fields.
// This query returns all elements in the database set at path
// `/dbname/complex` where
// - The field `x` of the `stored` record on field `s` of database set
//   elements is smaller that 100
// - and the intmap `imap` contains an elements associated to the key `10`
//   where the field `v` is equal to `"value"`
// - and the stringmap `smap` contains a bindings with the key `"key"`
dbset(_, _) x = /dbname/complex[s.x < 100 and imap[10].v == "value" and smap["key"] exists]
```

With hole expressions you can query elements within a list.

```
// Querying using hole expressions.
// This query returns all elements for which the stringmap `smap` contains a
// value associated to the `"key"` and the list on the field `lst` contains a
// value equal to "value".
dbset(_, _) x = /dbname/complex[smap["key"].lst[_] == "value"]
```

Updating

Previously we saw how to overwrite a path, like that :

```
/path/to/data <- x
```

which assigns to path `/path/to/data` the value of `x`. This used to be the only way to write to a path, but since Opa 0.9.0 (S4) there are more ways to update them.

Int updating

```
// Overwrite
/my_db/basic/i <- 42

// Increment
/my_db/basic/i++
/dbname/i += 10
```

```
// Decrement
/my_db/basic/i--
/my_db/basic/i -= 10

// Sure we can go across records
/my_db/r/x++
```

Record updating

```
// Overwrite an entire record
x = {x: 1, y: 2, v: "Hello, world", lst: []}
/my_db/r <- x

// Update a subset of record fields
/my_db/r <- {x++, y--}

/my_db/r <- {x++, v : "I just want to update z and incr x"}
```

List updating

```
// Overwrite an entire list
/my_db/l <- ["Hello", ",", "world", "."]

// Removes first element of a list
/my_db/l pop

// Removes last element of a list
/my_db/l shift

// Append an element
/my_db/l <+ "element"

// Append several elements
/my_db/l <++ ["I", "love", "Opa"]
```

Set/Map updating

The values stored inside database sets and maps can be updated as above. The difference is how we access the elements (more details in the querying section). Furthermore updates can operates on several paths.

```
//Increment the field y of the record stored at position 9 of imap
/dbname/imap[9] <- {y++}
```

```
//Increment fields y of records stored with key smaller than 9
/dbname/imap[< 9] <- {y++}

//Increment the field y of record where the field x is equals 9
/dbname/set[x == 9] <- {y++}

//Increment the field y of all records where the field x is smaller than 9
/dbname/set[x < 9] <- {y++}
```

Updating sub-paths

With sub-paths it is possible to update some values deeper in the database structure, relative to the point of update.

For instance, with indexed-expressions it's possible to write:

```
// Updating using expression fields.
// Update elements which match the query ‘...‘ by:
// - Incrementing field ‘x‘ of the ‘stored‘ record on field ‘s‘ of the database set
// - and add 2 to the field ‘x‘ and substract 2 from the field ‘y‘ of the
//   element associated to the ‘"key"‘ in the stringmap ‘smap‘
/dbname/complex[...] <- {s.x++, smap["key"].{x += 2, y -= 2}}
```

With hole expressions it is possible to update *the first* element of a list matched by the corresponding hole in the query.

```
// Updating using hole fields.
// Write "another" into the first element of the list ‘lst‘ in the map ‘smap‘
// at key "key".
/dbname/complex[smap["key"].lst[_] == "value"] <- {smap["key"].lst[_] : "another"}
```

Projection

We saw how we can query database sets and maps to obtain a sub-set of their values. But often we do not need all the information stored there but only a selected few fields and fetching complete information from the database would be inefficient.

We already saw how we can access a subpath of a given path:

```
// Access to /my_db/r
stored x = /my_db/r
// Access to the /x sub-path of /my_db/r
int x = /my_db/r/x
```

If we want to access to both subpath /x and /y of path /my_db/r we can write something like that:

```
{int x, int y} xy = {x : /my_db/r/x, y : /my_db/r/y}
```

But there is a more concise syntax to achieve the same:

```
// This path access means 'select fields x and y in path /my_db/r'
{int x, int y} x = /my_db/r.{x, y}
```

Similarly, you can use these kinds of projections on database sets and maps. Some examples follow:

```
// Access of /v sub-path of a set access with primary key
string result = /dbname/set[x == 10]/v

// Access of both sub-path /x and /v of a set access with primary key
{int x, string v} result = /dbname/set[x == 10].{x, v}

// Access of /v sub-paths of a set access
dbset(string, _) results = /dbname/set[x > 10, y > 0, v in ["foo", "bar"]]/v

// Access of two sub-paths /x and /v of a set access
dbset({int x, string v}, _) results = /dbname/set[x > 10, y > 0, v in ["foo", "bar"]].{x, v
```

You can also use sub-paths to project selected data.

```
dbset({{int x, int y} s, intmap(stored) imap}) results = /dbname/complex[...].{s.{x, y}, ima

dbset({{int x, int y} s, stored imap}) results = /dbname/complex[...].{s.{x, y}, imap["key"]
```

Manipulating paths

Most operations deal with data, as described above. We will now describe operations that deal with meta-data, that is the paths themselves.

A path written with notation `!/path/to/data` is called a *value path*. Value paths represent a snapshot of the data stored at that path. Should you write a new value at this path at a later stage, this will not affect the data or meta-data of such snapshots.

```
// Getting a value path. i.e. a read only path
!/path/to/data
```

Should you need to make reference to the *latest version* of data and meta-data, you will need to use a *reference path*, as obtained with the following notation:

```
// Getting a reference path. i.e. a r/w path
@/path/to/data
```

A path value is specific to its own database backend, indeed the type of a path contains 2 type variables; the first one for the type of data stored in the path and the second one for specifying the database backends used for this path.

For instance if `my_db` is declared as a MongoDb database, here is type of its paths :

```
Db.val_path(string, DbMongo.engine) !/my_db/s
Db.ref_path(string, DbMongo.engine) @/my_db/s
```

Generic functions to manipulate such values are available in the `Db` module (imported by default). There are also more functions specific to different backends: - `Db3` module from package `stdlib.database.db3` for Db3 backend features, such as path history. - `DbMongo` from package `stdlib.database.mongo` for MongoDb backend features.

Note, that the same mechanism is used for `dbset('data, 'engine)`

Db3 backend specifics

Restrictions :

- Most of querying/updating are not yet implemented

Db3 Transactions

Whenever database paths can be accessed simultaneously by many users, consistency needs to be ensured. For this purpose, Opa offers a mechanism of *transactions*. Apart from consistency it is also more efficient to explicitly encapsulate a sequence of database operations occurring in a row within a single transaction.

In their simplest and most common form, transactions take the form of function `Db.transaction`:

```
function atomic() {
    //...any operations
    void
}

result = Db.transaction(atomic)
match (result) {
 case {none}: //a conflict or another database error prevented the commit
 case ~{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 does not 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()

tr.in(atomic)

// some other treatments

match (tr.commit()) {
 case {success}: // ...
 case {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 from 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.

// Note: conflict resolution // At the time being, a transaction will conflict whenever some data that it writes // has been changed in the meantime. Other conflict policies are planned and, in the future, // it will be possible to select them on specific database paths (eg. conflict if // the transaction *read* some data that has been changed at the time of commit, // solve conflicts on a counter by adding the increments, etc.)

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.

Database Schema

Db3 verify that the database stored to the disk is compatible with the application. If the database schema has changed, Opa will offer the possibility to perform an automated database migration.

MongoDb backend specifics

Currently, the MongoDb driver does not verify if the compiled application is compatible with a database schema (the way the Db3 does). That means that after changing data used in the application the developer herself needs to take care of migrating the data.

Low-level MongoDB support

{block}[WARNING] This chapter is about advanced uses of MongoDB in Opa and details low-level access to MongoDB in Opa. For most applications, you should only read this chapter instead. {block}

Introduction

In this chapter, we describe the current state of support for MongoDB in the Opa standard library. We assume some familiarity with MongoDB concepts and particularly with the MongoDB shell. This familiarization can be gained by reading the MongoDB tutorial.

MongoDB is a server-based document-oriented non-relational database intended to be scalable and fast. Documents are stored in a binary JSON-like format called BSON. Although BSON has a richer set of types than JSON it is 100% compatible with JSON. For speed, MongoDB does not implement joins but is instead provided with a powerful query language of its own and almost anything that can be done with a relational database can be implemented in MongoDB with a little bit of effort (see MongoDB's page on SQL compatibility).

In addition, MongoDB allows multiple indices into its data although these are not automatic and have to be initiated in client code. MongoDB is intended to be deployed in reliable large-scale web-based applications and thus has features which facilitate scalability such as sharding and master-slave arrangements of servers along with features for reliability such as replicated servers with fail-over.

Backups of MongoDB data are usually done either offline on a slave server in the network using external tools or to redundant nodes in the MongoDB server network.

Setting-up MongoDB

If you are not familiar with the MongoDB database, here are some quick instructions to get you going. Firstly, make sure that you have MongoDB installed on your system:

```
% which mongod
```

Note that MongoDB doesn't come with any major packages such as Ubuntu, yet, but installation is trivial, download the latest version from the MongoDB downloads site and unpack the files locally. You should then just have to add the `bin` directory to your path and you should be up and running.

To run a MongoDB server, you first have to create a directory to store the database files. In fact, you need a directory for each node you wish to run, see the MongoDB documentation for how to create replica sets, sharding etc. At its simplest, start a `mongod` server with:

```
% mkdir -p ~/mongodata/master
% mongod --rest --oplogSize 500 --noprealloc --master --dbpath ~/mongodata/master > ~/mongoc
```

Use the `--oplogsize` and `--noprealloc` options to limit the initial allocated disk space (the default is about 1Gb). The `--rest` option allows you to monitor your database via the http interface (found at the port number plus 1000). If you wish to run the server on a different port, use the `--port 27017` option, the default MongoDB server port is 27017. Note, however, that to run the MongoDB shell on a non-default port you also need the `--port` option:

```
% mongo --port 27017
MongoDB shell version: 2.0.1
connecting to: test
>
```

For the MongoDB OPA drivers we recommend version 1.6.0 or greater since much of the current functionality was mature by that version. We always recommend the current MongoDB stable version (at the time of writing 2.0.2) but for the most part the driver is quite stable with respect to backwards compatibility.

Overview

The Opa support for MongoDB consists of a hierarchy of modules leading to successively higher-level programming.

Bson Support for the BSON binary format is in the form of the `Bson` module, all other modules are built on top of this one. In general, BSON values are handled by the `Mongo.document` Opa data-type but we also provide the `Bson.opa2doc` and `Bson.doc2opa` functions to allow conversion between Opa types and BSON documents.

MongoCommon This contains general support routines for dealing with replies from the MongoDB server. These include:

- printing results to meaningful strings
- testing results for error status
- handling tag lists instead of bit-mapped integers
- extracting fields and Opa types from MongoDB replies

MongoConnection The code which talks to the MongoDB server is in the private `MongoDriver` module. This includes support for replica sets with automatic reconnection on fail-over and cursors but for programming at this level we provide a single all-purpose module called `MongoConnection`.

Advanced programmers wishing to use some of the more obscure features of MongoDB can use the driver code directly but this is not recommended. MongoDB has a complex API involving over 70 functions and many of the simple access commands have numerous options. Our intention with this driver is to make accessing MongoDB databases as simple and logical as possible while still exposing the power and flexibility of the MongoDB engine.

MongoCommands As an adjunct to the low-level programming interface we provide a module containing a large (but still incomplete) number of the MongoDB command set called `MongoCommands`. These encompass most functions that will be required for meta-programming the MongoDB database, such as `dropDatabase`, `repairDatabase`, `createCollection` and so on plus functions associated with normal database access operations such as `getLastError`. The more advanced MongoDB functionality is also supported here, including `findAndModify` and the very powerful `mapReduce` function.

These commands occur in two flavors, those which return `Bson.document` values and those which convert their results into Opa types. If you are only looking for a single value out of a large and complex reply document then using the `Bson` module access functions on the raw BSON may be more efficient. If you intend complex analysis of the reply then the Opa types may be more convenient. At the present time only partial support is provided for Opa types. Some command results may never be treated this way because they include arbitrary field names which we can't safely convert into Opa types.

MongoCollection This module represents a type-safe view of the low-level routines in `MongoConnection`. Here, we insist upon Opa types as arguments and results from MongoDB operations. This necessarily limits what we can put into the database since the BSON documents stored in the database have to be consistent with the Opa types they represent.

To achieve this, we have implemented the `MongoSelect` and `MongoUpdate` modules which enforce a type discipline upon the arguments to, for example, `MongoCollection.insert`. The type safety is implemented as run-time type checks so there is a significant performance penalty for using these routines. In the future, however, we will provide fully type-safe compile-time type checks along the lines of the Opa internal database.

Programming

Here, we provide some notes on programming with the Opa MongoDB driver. The full interface is too large for complete coverage here, refer to the online Opa API documentation for detailed notes on each function.

Using BSON types in Opa

The full Opa BSON data-type is as follows:

```
/**
 * A BSON value encapsulates the types used by MongoDB.
 **/
```

```
type Bson.value =
    { float Double }
 or { string String }
 or { Bson.document Document }
 or { Bson.document Array }
 or { string Binary }
 or { string ObjectID }
 or { bool Boolean }
 or { Date.date Date }
 or { Null }
 or { (string, string) Regexp }
 or { string Code }
 or { string Symbol }
 or { (string, Bson.document) CodeScope }
 or { int Int32 }
 or { int32 RealInt32 }
 or { (int, int) Timestamp }
 or { int Int64 }
 or { int64 RealInt64 }
 or { Min }
 or { Max }

/**
 * A BSON element is a named value.
 **/
type Bson.element = { string name, Bson.value value }

/**
 * The main exported type, a BSON document is just a list of elements.
 */
type Bson.document = list(Bson.element)
```

While values of this type can be constructed manually:

```
doc = Bson.document
      [{name: "$eval", value: {Code:"function(x,y) \{return x*y;}"}},
       {name: "args", value:{Array:[{name:"0", value:{Int32:6}},
                                    {name:"1", value:{Int32:7}}]}}]
```

there are two more convenient ways of constructing BSON values. Firstly, we provide a set of abbreviations in the `Bson.Abbrevs` module:

```
H = Bson.Abbrevs
doc = Bson.document [H.code("$eval","function(x,y) \{return x*y;}"),
                     H.valarr("args",[{Int32:6},{Int32:7}])]
```

Secondly, we can construct the values in Opa and use `Bson.opa2doc`:

```
doc = Bson.opa2doc({`$eval`:(Bson.code "function(x,y) \{return x*y;}"),
                    args:(list(Bson.int32) [6,7])})
```

Notice that to get a field with non-alphanumeric characters we have to backquote the field name in the Opa value and that to control the representation in the BSON type we can apply helper types, for example `Bson.code` is just a string but it instructs `Bson.opa2doc` to treat it as code. Remember also to escape curly brackets in strings. Note that to get `Int32` values you need the `Bson.int32` type, the default for `int` is actually `Bson.int64`.

There are several such types provided by the `Bson` module but some merit special mention:

- Optional types have a special significance with respect to `Bson.doc2opa` in that if a field value is missing in the document it will appear in the Opa type as {`none`}. The alternate direction does not apply, {`none`} values are represented in the BSON document as { `none : null` }.

```
type Bson.register('a) = {'a present} or {absent}
```

- We take this one step further, however, with the `Bson.register` type, which actually behaves much as `option('a)` except that when we call `Bson.doc2opa` any {`absent`} values are omitted from the resulting document altogether. Note that there is a module `Bson.Register` which provides the same functionality for `Bson.register` as the `Option` module does for type `option`.

- Care should be taken in dealing with integer values which may have been placed into the database outside of OPA. OPA uses, internally, the OCaml integer representation `int` which is actually 31 bits wide on 32-bit systems and 63 bits wide on 64-bit systems (the spare bit is reserved by the garbage collector). Now MongoDB actually uses fully 32-bit and 64-bit integers which means that it is possible to find an integer value in a MongoDB database which is too large for the OPA representation (remember that all values generated by OPA and stored in the database are guaranteed to be within range). Currently, OPA only has 32-bit and 64-bit integers as abstract values. Such values can be stored in OPA as an external type (`int32` and `int64`) but no operations are possible on these values (they are sometimes needed by external libraries). We handle this situation in the MongoDB driver by automatically detecting overflow values and storing them as `RealInt32` and `RealInt64` when returning `Bson.document` types from the driver. While these values may appear to be invisible to the `Bson` module functions such as `find_int`, you can detect overflows by inspecting the document values:

```
match (value) {
  case {RealInt32:_}: error("overflow");
  case {Int32:i}: i;
  default: error("not an int");
}
```

- The `Bson.meta` type is intended to support situations where MongoDB can return a field of different types depending upon the nature of the command executed. A good example of this is the `out` option to the `mapReduce` function which can be either a `string` or a document type. We cast the parameter as `Bson.meta` which allows us to control the type at the function's application. We can also apply this trick to the `result` type from `mapReduce` calls:

```
mr = MC.mapReduceSimple(mongodb,map,reduce,{String:"example1"})

/* or */

mr = MC.mapReduceSimple(mongodb,map,reduce,{Document:[H.str("reduce","session_stat")]})
```

- Two other cases should be mentioned. Both `list` and `intmap` are mapped onto `Array` values in BSON. The difference is that `list` is mapped to consecutive-numbered elements in the `Array` document whereas `intmap` allows sparse arrays.

As a rough guide to `Bson.opa2doc` and `Bson.doc2opa`, the following simple schema shows the mapping:

```
/* We use a "natural" mapping of constant types */
float <-> Double
string <-> String
Bson.binary <-> Binary
Bson.oid <-> ObjectID
bool <-> Boolean
Date.date <-> Date
void <-> Null
Bson.regexp <-> Regexp
Bson.code <-> Code
Bson.symbol <-> Symbol
Bson.codescope <-> CodeScope
Bson.int32 <-> Int32
Bson.realint32 <-> Int32
Bson.timestamp <-> Timestamp
Bson.realint64 <-> Int64
Bson.min <-> Min
Bson.max <-> Max

 /* Basic record scheme */
{a:'a; b:'b} <-> { a: 'a, b: 'b }

 /* Sum types */
{a:'a} / {b:'b} <-> { a: 'a } <or> { b: 'b }

 /* Non-record types are called "value" */
'a <-> { value: 'a }

 /* Special cases */

 /* Default for int is Int64 */
int <-> Int64

 /* Overflow */
Bson.realint32 <- Int32 /* when integer exceeds range */
Bson.realint64 <- Int64 /* when integer exceeds range */

 /* Options */
option('a):
  {some=a} <-> { some : 'a }
  {none} <-> { none : null }
  {none} <- { }

 /* Registers */
Bson.register('a):
  {present=a} <-> { present : 'a }
  {absent} <- { absent : null }
  {absent} <-> { }

 /* Lists are consecutive arrays */
list('a) <-> { Array=(<label>,{ 0:'a; 1:'a; ... }) }

 /* Intmaps are non-consecutive arrays */
ordered_map(int,'a) <or>
intmap('a) <-> { Array=(<label>,{ 1:'a; 3:'a; ... }) }

 /* Bson.document is treated verbatim (including labels) */
Bson.document <-> Bson.document

 /* Bson.meta is treated as a variable type */
int:Bson.meta <-> { Int64:int }
string:Bson.meta <-> { String:string }
```

```
bool:Bson.meta <-> { Boolean:bool }
etc.
```

Notes:

- For `ObjectID` values, there are a couple of routines which convert between (hex value) strings and the BSON representation, `Bson.oid_of_string` and `Bson.oid_to_string`. You can also create a BSON-style OID value with `Bson.new_oid`.
- `Bson.document` types are completely write-through, i.e. they are not processed at all.
- In case you're wondering, `Min` and `Max` are used in sharded databases to indicate infimum and supremum bounds on sharding regions, respectively.

//TODO: other functions find_xyz, to_pretty, error stuff

Using the low-level interface

Connecting to and using the low-level drivers should be done using the `MongoConnection` module. This gathers together various low-level features in a single module.

Opening a connection to the MongoDB server The preferred method is to use the system of named connections which can be defined from the command line or setup internally using the `Mongo.param` type and the `MongoConnection.add_named_connection` function.

Initially, there is one default connection (called ''default'') which is set to `localhost:27017`, the default port for MongoDB servers on the local machine. To open this connection use:

```
mongodb =
  match (MongoConnection.open("default")) {
    case {success:mongodb}: mongodb
    case {~failure}: ... /* take action on error */
  }

/* or */

mongodb = MongoConnection.openfatal("default")
```

The `MongoConnection.open` function returns an outcome of either the connection or the standard `Mongo.failure` type whereas the `MongoConnection.openfatal` function returns just the connection but treats a failed connection as a fatal error.

To setup the connection from the command line the following options are defined:

{table} {* Option | Abbrev Type | Description *} {| `--mongo-name` | `(--mn)` <`string`> | Name for the MongoDB server connection |} {| `--mongo-repl-name` | `(--mr)` <`string`> | Replica set name for the MongoDB server |} {| `--mongo-buf-size` | `(--mb)` <`int`> | Hint for initial MongoDB connection buffer size |} {| `--mongo-socket-pool`| `(--mp)` <`int`> | Number of sockets in socket pool (>=2 enables socket pool) |} {| `--mongo-seed` | `(--ms)` <`host`>{`:`<`port`>} | Add a seed to a replica set, allows multiple seeds |} {| `--mongo-host` | `(--mh)` <`host`>{`:`<`port`>} | Host name of a MongoDB server, overwrites any previous hosts |} {| `--mongo-log` | `(--ml)` <`bool`> | Enable MongoLog logging |} {| `--mongo-log-type` | `(--mt)` <`string`> | Type of logging: stdout, stderr, logger, none |} {| `--mongo-auth` | `(--ma)` <`user:pwd@dbname`> | Define user name and password for database dbname |} {table}

So, for example, to connect to the default connection at `machinexyz:12345` you would use:

```
% prog.exe --mh machinexyz:12345
```

This remains a single connection, to connect to a replica set you also need to define a name for the replica set plus some seeds:

```
% prog.exe --mn blort --mr blort --ms machinexyz:27017 --ms machineuvw:27017
```

Here we have defined a connection called ''blort" to a replica set also called ''blort" with two seed machines. Remember that you only really need one seed which is active in the set, the connection logic queries the seeds for the actual host list and then polls the hosts until it finds the current primary server. From then on reconnection will be attempted if the current primary goes down.

Note that you can define as many named connections as you like, this example still retains the default connection.

Note also that you can clone a connection such that the connection itself will not be closed until all clones have already been closed.

Handling concurrency within an Opa program is done by a socket pool. This means that a pool of open connections is maintained to the same server such that blocking only occurs if there are no more available connections in the pool (set with `--mp 2`, for example). If you ensure that the pool size is at least as big as the number of threads in your code then no blocking will occur.

Named connections can also be defined within the program:

```
MongoConnection.add_named_connection({
  name: "blort",
  replname: {some: "blort"},
  bufsize: 50*1024,
  pool_max: 2,
  log: false,
  seeds:[("localhost",10001),("localhost",10002)],
  auth:[{dbname:"mydb",user:"me",password:"secret"}]
})

mongodb2 = N.openfatal("blort")
```

Once a connection has been opened, it can be pointed to different databases and collections using a functional interface. The default database is ''db" and the default collection is ''collection" but we can make a connection to a different collection without re-opening the connection as follows:

```
mongodb_wiki = MongoConnection.namespace(mongodb,"db","wiki")
```

This mechanism also applies to the flags that some of the MongoDB operations can take, for example to set the `Upsert` flag for all insert operations:

```
mongodb3 = MongoConnection.upsert(mongodb)
```

This method is quite flexible since you can define these flags once when the connection is made, making the flags globally persistent, or you can add these function calls at the point of calling the operation, i.e. locally defined flags (there are examples below). All of the MongoDB flags are supported in this way.

One particular flag is worth mentioning, the `log` flag which can be set on the command line and can actually be overridden in this way allowing you to generate logs for targeted sections of code. In fact, you can change any of the command line options this way but bear in mind that some of them, for example, seed lists, will not take effect until the connection is reconnected.

Authentication As you can see, you can add the MongoDB authentication parameters for a given database either on the command line using the `--mongo-auth` argument which is of the form: `user:password@database_name` or by placing the authentication parameters in the `auth` field in the `add_named_connection` function argument.

Alternatively, you can call the `MongoCommands.authenticate` function to perform an additional, external authentication. Note that if you are connecting to a replica set then the driver needs to re-authenticate after connecting to the new host so the authentication parameters are built into the low-level Mongo datatype. This means that if you call this function you should perform all subsequent operations on the returned Mongo datatype, not on the original which won't have the parameters built in.

Remember that authentication in MongoDB is to a database, not to a connection so you can have multiple user names and passwords associated with a single connection. If you want to authenticate with all of the databases over a connection you need to authenticate with the `admin` database which acts a bit like ''root" access for databases.

Basic operations The basic database access operations are the same as the MongoDB protocol operations, i.e. insert, update, query, get_more, delete, kill_cursors and msg. So, for example, to insert a document:

```
/* A couple of documents */
p1 = [H.str("name","Joe1"), H.i32("age",44)]
p2 = [H.str("name","Joe2"), H.i32("age",55)]

/* Insert the documents */
MongoConnection.insert(mongodb,p1)
MongoConnection.insert_batch(mongodb,[p1,p2])
```

The basic write operations come in three types:

- `insert` is the write-and-forget operation where the insert message is sent and a boolean value is returned which simply states that the correct number of bytes were written to the socket.
- `inserte` is a ''safe" operation where the insert message has a `getlasterror` query piggy-backed onto it and then the raw optional reply is returned.
- `insert_result` does an `inserte` and then analyzes the reply, turning it into a standard `Mongo.result` type.

All of the basic write operations have these three forms. The `Mongo.result` type is an `outcome` of either success as a `Bson.document` type or failure as a `Mongo.failure` type. The `Mongo.failure` type looks like:

```
type Mongo.failure =
    {OK}
```

```
 or {string Error}
 or {Bson.document DocError}
 or {Incomplete}
 or {NotFound}
```

This defines either a raw document error {`DocError:doc`} which is an error as reported by the MongoDB server, a driver error {`Error:str`} which is a message generated by the Opa driver or a few special-purpose errors returned under specific circumstances ({`OK`} is simply a connection that has never been used).

Post-processing of results may include checking for errors:

```
error = MongoConnection.insert_result(MongoConnection.upsert(mongodb),[H.i32("i",n)])
println("insert error={MongoCommon.is_error(error)}")
```

or extracting specific fields from the reply:

```
println("errmsg={MongoCommon.result_string(error,"errmsg")}")
```

noting that we also support the MongoDB dot notation syntax:

```
println("indexSizes._id_={MongoCommon.dotresult_int(collStats,"indexSizes._id_")}")
```

Closing a connection is as simple as:

```
MongoConnection.close(mongodb)
```

Remember that the connection will only close once all of the clones have also been closed.

Cursors Handling queries in MongoDB has the complication that, for efficiency, cursors are stored on the server which entails tracking them at the client side. While the bare `MongoConnection.query` and `MongoConnection.get_more` operations can be used to handle queries in conjunction with the reply support code in `MongoCommon` they are a bit inconvenient.

For this purpose we have defined cursor operations in the `MongoCursor` module and re-exported the most important ones into the `MongoConnection.Cursor` module. A cursor object itself contains all the parameters needed to manage the cursor at the server side and, in fact, duplicates some of the information in the connection object. Using the re-exported functions reduces the number of parameters to the basic functions since this information can be lifted from the connection into the cursor object.

Here is an example of a low-level cursor dialog:

```
cursor = MongoConnection.Cursor.init(mongodb)
cursor = MongoConnection.Cursor.set_query(cursor,{some:[H.str("name","Joe")]})
cursor = MongoConnection.Cursor.set_limit(cursor,3)
cursor = MongoConnection.Cursor.set_fields(cursor,{some:[H.i32("_id",0)]})
cursor = MongoConnection.Cursor.next(cursor)
result = MongoConnection.Cursor.check_cursor_error(cursor)
println("result 1 = {MongoCommon.pretty_of_result(result)}")
println("valid 1 ={MongoConnection.Cursor.valid(cursor)}")
cursor = MongoConnection.Cursor.next(cursor)
result = MongoConnection.Cursor.check_cursor_error(cursor)
println("result 2 = {MongoCommon.pretty_of_result(result)}")
println("valid 2 = {MongoConnection.Cursor.valid(cursor)}")
MongoConnection.Cursor.reset(cursor)
```

The cursor is initialized with `init` and then the parameters for the query are setup. The `next` function generates the `query` (or `get_more`) call to the server and places the next document internally in the cursor object along with any error status. The `check_cursor_error` function is a convenient way of extracting either the current document or the error as a `Mongo.result`. Subsequent calls to `next` will either return the next document from the previous reply or issue a `get_more` call to re-populate the cursor. The end of the matching documents (or if no document matches) is signaled with `NotFound` and if you try to read past the end of matching documents you will get an ''end of data" error from the driver. The `valid` function is used to poll whether there is any remaining data. Finally, the call to `reset` is important here because it doesn't just end the query, it will issue a `kill_cursors` operation to the server to tell it to delete the cursor (cursors time out after 10 minutes by default on the MongoDB server).

This method works fine but this logic has been wrapped up into some convenience functions:

- `find_one` returns the first matching document as a `Mongo.result`
- `find_all` gives all the matches as a list of documents (use the `limit` function to limit the number of replies).

For example:

```
/* Find all objects in db.session, excluding the _id field */
mongo_session_no_id =
  MongoConnection.fields(MongoConnection.namespace(mongodb,"db","session"),{some:[H.i32("_ic
println("findAll: {CM.pretty_of_results(MongoConnection.Cursor.find_all(mongo_session_no_id,
```

You can also define custom loops over the matches using `start` (or `find`) in conjunction with `next` and `valid`. (Note that you must use the

`MongoConnection.Cursor.for` loop instead of the more usual `for` function in the Opa stdlib, you need to check for valid and only call next if still valid at that point, otherwise you will miss the last document in the list of matches).

//Commands //˜˜˜˜˜˜˜˜

Collections

While you can achieve anything that MongoDB is capable of using the low-level drivers, there are no guarantees of type safety while converting between BSON documents and Opa values. You can of course base your entire project around BSON values and eliminate the need for converting between MongoDB's documents and Opa types altogether but this may not be very convenient depending upon what is happening elsewhere in your application. Secondly, to use the low-level drivers requires an investment in learning MongoDB's powerful but rather complex interface (which may be new to users of relational databases) in order to exploit what MongoDB has to offer. Finally, basing your application on MongoDB's API will tie your application to MongoDB and you may at some point in the future wish to migrate to other database solutions.

Ultimately, the intention is to provide an abstract view of the database which is general enough to encompass several of the existing database solutions, of which MongoDB is an important player, and support this with compiler-generated syntax in the manner of the Opa inbuilt database. This support is still not available but we can offer an intermediate layer of programming MongoDB whereby we assume collections of Opa types and support type-safety by performing run-time type-checks on operations over these collections. This support is in the form of the `MongoCollection` module plus some support modules for generating values suitable to be applied to these functions.

The `collection` type

The central idea in the `MongoCollection` module is a collection (in the MongoDB terminology sense) of Opa values. This is embodied in the `Mongo.collection` type which is extremely simple, it's just a `MongoConnection` value cast to the specific type of the values to be stored in the collection:

```
type Mongo.collection('a) = {
  Mongo.mongodb db /* the mongodb connection */
}
```

When a value is stored in the collection it is automatically converted from its Opa type into a matching BSON document and *vice versa* for queries.

While this sounds simple there are a number of pitfalls to watch out for. We assume that any offline modifications of the collection will not create any incompatible values. If, for example, we add or delete a field from a record then the entry can no longer be represented as an Opa type.

To overcome this problem we place checks in the code to verify the suitability of documents read from the collection and an error will be generated if any such values are found. We also provide features to allow handling of this situation in some specific circumstances, for example, if you type a field in the collection as `Bson.register` it will allow you to successfully read in values with missing fields but this is not recommended for collections. Ultimately, it is up to the maintainer of the database to ensure that the values stored there are consistent with the application's usage of the collection.

Despite these provisos, using a collection is very simple and gives the programmer the ability to integrate Opa types with the MongoDB system without having to understand the underlying complexity of the database and with a modest level of type-safety. The cost, for the moment, is the overhead of the run-time type-checks which will slow down database operations.

Programming with collections

A simple dialog for creating and manipulating a collection might be as follows:

```
/* The type of our first collection */
type t = {int i}

/* Create a collection of type t */
Mongo.collection(t) c1 = MongoCollection.openfatal("default","db","collection")

/* Put a single value into the collection */
result = MongoCollection.insert_result(c1,{i:0})

/* Finally, destroy the collection */
MongoCollection.destroy(c1)
```

We define a type for the collection (`type t`) so that when we open a connection to the database we can cast the resulting collection object and thus install the correct run-time representation of the type. The `openfatal` function returns a collection and treats a connection failure as fatal. There are several variants of the `open` function.

A collection is a pointer to a specific collection in the database (here, `db.collection`) and we create a connection to the MongoDB server using the connection name (in this instance, `default`).

Inserting a value into the collection is trivial, the value is simply passed as it is to the `insert` function (here we use the safe `insert_result` function which also returns the result of a `getlasterror` call). The insert has exactly the same effect as a call to `MongoConnection.insert` but with the value automatically converted into a BSON document using the scheme outlined above.

The call to `MongoCollection.destroy` should not be forgotten because this closes the underlying connection.

While the `insert` function is trivial, we need more care with `update` and `delete`. The problem is that to maintain our level of type-safety we need to match select (and update) documents with the type of the collection they are applied to. We do this with a system of run-time type-checks applied to the select documents. For example:

```
/* Create pre-typed select and update generation functions */c
MongoSelect.create reatest = Bson.document -> Mongo.select(t)
MongoUpdate.create createut = Bson.document -> Mongo.update(t)

/* Generate the select documents */
select = createst(MongoSelectUpdate.int64(MongoSelectUpdate.empty(),"i",0))
update = createut(MongoSelectUpdate.inc(MongoSelectUpdate.int64(MongoSelectUpdate.empty(),"i

/* We can now apply update to these documents */
result = MongoCollection.update_result(c1,select,update)
```

Firstly, we use the `MongoSelectUpdate` module to generate the basic documents. Note that we could also have used the `Bson.opa2doc` function to achieve the same result:

```
select = createst(Bson.opa2doc({i:0}))
update = createut(Bson.opa2doc({‘$inc‘:{i:1}}))
```

The choice between these two styles may depend upon the type of document being generated. The Opa type-based versions are more readable but the `MongoSelectUpdate` ones are much faster since no conversion is required.

The select documents have to be correctly typed for the collection they apply to so we generate a couple of convenience functions `createst` and `createut` to do the casting for us.

Secondly, once we have these documents we can apply the `update` function to them but note that although a select document is just a typed `Bson.document` it triggers a set of suitability tests. These tests are complex and probably do not cover all possible MongoDB operations but briefly, the select document is scanned by a knowledge-base of the types of MongoDB field types, for example `$inc` only applies to updates, `$and` only applies to selects whereas `$comment` can apply to both. Once the status (select/update/both) is determined, the type of the resulting values is determined from the select document and is verified to be a subtype of the type of the collection. So, for example, {`int a`} is a subtype of {`int a, string b`} but {`int a, bool c`} is not. Presently, we only print a suitable warning but in future, once these routines have fully matured we may return an error value.

All of the basic database write operations occur in both send-and-forget and in send-with-getlasterror forms: `insert`, `insert_result`, `insert_batch`, `insert_batch_result`, `update`, `update_result`, `delete` and `delete_result`.

As an aside, notice that we use a similar functional interface for flags as for the low-level code:

```
MongoCollection.delete(MongoCollection.singleRemove(c1),createst(Bson.opa2doc({i:104})))
```

The select mechanism applies to queries as well but in this case we have to be careful what types we return from the database:

```
result = MongoCollection.find_one(c1,createst(Bson.op12doc({`$where`:(Bson.code "this.i > 1
match (result) {
  case {success:{~i}}: println("i={i}")
  case {~failure}: println("error={MongoCommon.string_of_failure(failure)}")
}
```

This example returns the first value in the collection for which `i` is greater than 106, it expresses the select as a Javascript expression. Many of the MongoDB query methods are perfectly safe with collections such as the `$where` example here but some methods are not safe in that they return documents which contain fields other than those in the Opa type, a good example being the http://www.mongodb.org/display/DOCS/Explain[`$explain`] documents which are a set of statistical data concerning the given query (see the `Mongo.explainType` type in `MongoCommands`). In general, we attempt to support such features with special purpose functions rather than via the normal database operations.

The usual simplified query functions are present in `MongoCollection`, `find_one` and `find_all`. There are also two functions which return the bare `Bson.document` representation of the result, `find_one_doc` and `find_all_doc` which may be useful in the above situation where the result of the query is not compatible with Opa types. For more general query scanning, the cursor-based routines are available. For example, the following code scans the results of a `MongoCollection` query

```
query = createst(Bson.opa2doc({i:{`$gt`:102, `$lt`:106}}))
match (MongoCollection.query(MongoCollection.limit(c1,0),query)) {
  case {success:cc1}:
    cc1 =
      while(cc1,(function(cc1) {
                   match (MongoCollection.next(cc1)) {
                     case (cc1,{success={~i}}):
                        println("i={v}")
                        (cc1,MongoCollection.has_more(cc1))
                     case (cc1,{~failure}):
                        println("error={MongoCommon.string_of_failure(failure)}")
                        (cc1,false))})
    MongoCollection.kill(cc1)
  case {~failure}:
    println("error={MongoCommon.string_of_failure(failure)}")
}
```

In this code, we create a `Mongo.collection_cursor` object using `MongoCollection.query` to which we can then apply the collection-specific cursor functions `MongoCollection.next` and `MongoCollection.has_more`. This allows arbitrary processing of collection queries. Remember, as with the low-level cursors above, that the `MongoCollection.kill` function does not just end the scan, it also sends a `kill_cursors` message to the MongoDB server to tell it to destroy the cursor.

Another aside in this code is that we set the `limit` value to `0` which means ''use the default number of documents per reply". If we had set this to `1` we would only ever get one document in the reply because MongoDB treats this as a special case, i.e. ''just return one document".

Again, to help with the situation where return values may be incompatible with Opa types, we provide the `_unsafe` variants of the query functions. These, for example `query_unsafe`, take an additional boolean flag, `ignore_incomplete` which instructs the driver to simply ignore any return documents which have missing fields and are thus not compatible with Opa types. MongoDB will actually return partial documents if the document meets the query document but does not contain all of the fields (an exception is the `_id` field which is always returned unless specifically excluded with the return field selector document). These functions should be used with care.

Apart from the support described here the `MongoCollection` module also provides a few convenience functions such as creating indexes using collection objects and some direct support for some of the aggregation functions (`count`, `distinct` and `group`). Finally, one of the variants of the `open` function, `openpkg` and `openpkgfatal` supplies a set of pre-cast versions of `MongoSelect.create` and `MongoUpdate.create`.

Example: Hello, MongoDB wiki

In this section, we describe how to convert the `hello_wiki` example described in the previous chapter to using the MongoDB database. This is actually a simple process and uses MongoDB as a simple key-value storage database.

// TODO: more realistic example

The first task is to open a connection to the database. We are going to use collections and in fact, we will use the version of `open` which also gives us the casting functions for selects:

```
/**
 * The basic info. about the database and table location.
 */
type page = {
  string _id,
  Bson.int32 _rev,
  string content
}

/**
 * We work at level 1, run-time type-checked storage of a collection of Opa values.
 * The Mongo.pkg type provides convenience functions for building select and update documen
 **/
Mongo.pkg(page) (wiki_collection,wiki_pkg) = MongoCollection.openpkgfatal("default","db","wi
function pageselect(v) { wiki_pkg.select(Bson.opa2doc(v)); }
function pageupdate(v) { wiki_pkg.update(Bson.opa2doc(v)); }
```

The `_rev` field has been cast to `Bson.int32` so we can use 32-bit integers for this field (it is unlikely we will ever have more than 4 giga-revisions of any value in the database!). We then open our connection using the default named connection and connect to the collection `db.wiki`. This returns a collection object plus a package of values which we use to build our select documents.

Next we are actually going to search for documents including the `_rev` field so we can't just use the default index for our collection (the `_id` field):

```
/**
 * Indexes aren't automatic in MongoDB apart from the non-removable _id index.
 * Since we're searching on _rev as well, we need a separate index.
 **/
MongoCollection.create_index(wiki_collection, "db.wiki", Bson.opa2doc({_id:1; _rev:1}), 0)
```

The `get_content` function can then be modified using a simple call to `MongoCollection.find_one`:

```
function get_content(docid) {
  default_page = "This page is empty. Double-click to edit."
  function extract_content(page record) { record.content }
  /* Order by reverse _rev to get highest numbered _rev. */
  orderby = {some:Bson.opa2doc({_rev:-1})}
  match (MongoCollection.find_one(MongoCollection.orderby(wiki_collection,orderby),pageselec
    case {success:page}: extract_content(page)
    case {failure:{NotFound}}: default_page
    case {~failure}:
      jlog("hello_wiki_mongo: failure={MongoCommon.string_of_failure(failure)}")
      default_page
  }
}
```

We search the database for the given _id value but we want the highest-numbered _rev field so we sort by inverse order on that field (the default ordering for numerical fields is in increasing order). A missing document is signaled by the NotFound failure condition, other failure values are errors.

Finally, the save_source function becomes a call to MongoCollection.update_result:

```
exposed function save_source(topic, source) {
  select = pageselect({_id:topic})
  update = pageupdate({`$set`:{content:source}, `$inc`:{_rev:(Bson.int32 1)}})
  /* Upsert this so we create it if it isn't there */
  result = MongoCollection.update_result(MongoCollection.upsert(wiki_collection),select,upda
  if MongoCommon.is_error(result)
    then <>Error: {MongoCommon.pretty_of_result(result)}</>;
    else load_rendered(topic);
}
```

In this case, we select only the _id field and we update the document by setting the content field and incrementing the _rev field. Note that we use the Upsert flag which tells MongoDB to insert the document if it isn't already present in the collection. We test the result for errors using the safe update operation but apart from that the code is identical to the existing ''Hello wiki" example.

Low-level MongoDB support

{block}[WARNING] This chapter is about advanced uses of MongoDB in Opa and details low-level access to MongoDB in Opa. For most applications, you should only read this chapter instead. {block}

Introduction

In this chapter, we describe the current state of support for MongoDB in the Opa standard library. We assume some familiarity with MongoDB concepts and particularly with the MongoDB shell. This familiarization can be gained by reading the MongoDB tutorial.

MongoDB is a server-based document-oriented non-relational database intended to be scalable and fast. Documents are stored in a binary JSON-like format called BSON. Although BSON has a richer set of types than JSON it is 100% compatible with JSON. For speed, MongoDB does not implement joins but is instead provided with a powerful query language of its own and almost anything that can be done with a relational database can be implemented in MongoDB with a little bit of effort (see MongoDB's page on SQL compatibility).

In addition, MongoDB allows multiple indices into its data although these are not automatic and have to be initiated in client code. MongoDB is intended to be deployed in reliable large-scale web-based applications and thus has features which facilitate scalability such as sharding and master-slave arrangements of servers along with features for reliability such as replicated servers with fail-over.

Backups of MongoDB data are usually done either offline on a slave server in the network using external tools or to redundant nodes in the MongoDB server network.

Setting-up MongoDB

If you are not familiar with the MongoDB database, here are some quick instructions to get you going. Firstly, make sure that you have MongoDB installed on your system:

```
% which mongod
```

Note that MongoDB doesn't come with any major packages such as Ubuntu, yet, but installation is trivial, download the latest version from the MongoDB downloads site and unpack the files locally. You should then just have to add the `bin` directory to your path and you should be up and running.

To run a MongoDB server, you first have to create a directory to store the database files. In fact, you need a directory for each node you wish to run, see the MongoDB documentation for how to create replica sets, sharding etc. At its simplest, start a `mongod` server with:

```
% mkdir -p ~/mongodata/master
% mongod --rest --oplogSize 500 --noprealloc --master --dbpath ~/mongodata/master > ~/mongod
```

Use the `--oplogsize` and `--noprealloc` options to limit the initial allocated disk space (the default is about 1Gb). The `--rest` option allows you to monitor your database via the http interface (found at the port number plus 1000). If you wish to run the server on a different port, use the `--port 27017` option, the default MongoDB server port is 27017. Note, however, that to run the MongoDB shell on a non-default port you also need the `--port` option:

```
% mongo --port 27017
MongoDB shell version: 2.0.1
connecting to: test
>
```

For the MongoDB OPA drivers we recommend version 1.6.0 or greater since much of the current functionality was mature by that version. We always recommend the current MongoDB stable version (at the time of writing 2.0.2) but for the most part the driver is quite stable with respect to backwards compatibility.

Overview

The Opa support for MongoDB consists of a hierarchy of modules leading to successively higher-level programming.

Bson Support for the BSON binary format is in the form of the `Bson` module, all other modules are built on top of this one. In general, BSON values are handled by the `Mongo.document` Opa data-type but we also provide the `Bson.opa2doc` and `Bson.doc2opa` functions to allow conversion between Opa types and BSON documents.

MongoCommon This contains general support routines for dealing with replies from the MongoDB server. These include:

- printing results to meaningful strings
- testing results for error status
- handling tag lists instead of bit-mapped integers
- extracting fields and Opa types from MongoDB replies

MongoConnection The code which talks to the MongoDB server is in the private `MongoDriver` module. This includes support for replica sets with automatic reconnection on fail-over and cursors but for programming at this level we provide a single all-purpose module called `MongoConnection`.

Advanced programmers wishing to use some of the more obscure features of MongoDB can use the driver code directly but this is not recommended. MongoDB has a complex API involving over 70 functions and many of the simple access commands have numerous options. Our intention with this driver is to make accessing MongoDB databases as simple and logical as possible while still exposing the power and flexibility of the MongoDB engine.

MongoCommands As an adjunct to the low-level programming interface we provide a module containing a large (but still incomplete) number of the MongoDB command set called `MongoCommands`. These encompass most functions that will be required for meta-programming the MongoDB database, such as `dropDatabase`, `repairDatabase`, `createCollection` and so on plus functions associated with normal database access operations such as `getLastError`. The more advanced MongoDB functionality is also supported here, including `findAndModify` and the very powerful `mapReduce` function.

These commands occur in two flavors, those which return `Bson.document` values and those which convert their results into Opa types. If you are only looking for a single value out of a large and complex reply document then using the `Bson` module access functions on the raw BSON may be more efficient. If you intend complex analysis of the reply then the Opa types may be more convenient. At the present time only partial support is provided for Opa types. Some command results may never be treated this way because they include arbitrary field names which we can't safely convert into Opa types.

MongoCollection This module represents a type-safe view of the low-level routines in `MongoConnection`. Here, we insist upon Opa types as arguments and results from MongoDB operations. This necessarily limits what we can put into the database since the BSON documents stored in the database have to be consistent with the Opa types they represent.

To achieve this, we have implemented the `MongoSelect` and `MongoUpdate` modules which enforce a type discipline upon the arguments to, for example, `MongoCollection.insert`. The type safety is implemented as run-time type checks so there is a significant performance penalty for using these routines. In the future, however, we will provide fully type-safe compile-time type checks along the lines of the Opa internal database.

Programming

Here, we provide some notes on programming with the Opa MongoDB driver. The full interface is too large for complete coverage here, refer to the online Opa API documentation for detailed notes on each function.

Using BSON types in Opa

The full Opa BSON data-type is as follows:

```
/**
 * A BSON value encapsulates the types used by MongoDB.
 **/
type Bson.value =
    { float Double }
 or { string String }
 or { Bson.document Document }
 or { Bson.document Array }
 or { string Binary }
 or { string ObjectID }
 or { bool Boolean }
 or { Date.date Date }
 or { Null }
 or { (string, string) Regexp }
 or { string Code }
 or { string Symbol }
 or { (string, Bson.document) CodeScope }
 or { int Int32 }
 or { int32 RealInt32 }
 or { (int, int) Timestamp }
 or { int Int64 }
 or { int64 RealInt64 }
 or { Min }
 or { Max }

/**
 * A BSON element is a named value.
 **/
type Bson.element = { string name, Bson.value value }

/**
 * The main exported type, a BSON document is just a list of elements.
 */
type Bson.document = list(Bson.element)
```

While values of this type can be constructed manually:

```
doc = Bson.document
      [{name: "$eval", value: {Code:"function(x,y) \{return x*y;}"}},
       {name: "args", value:{Array:[{name:"0", value:{Int32:6}},
                                    {name:"1", value:{Int32:7}}]}}]
```

there are two more convenient ways of constructing BSON values. Firstly, we provide a set of abbreviations in the `Bson.Abbrevs` module:

```
H = Bson.Abbrevs
doc = Bson.document [H.code("$eval","function(x,y) \{return x*y;}"),
                     H.valarr("args",[{Int32:6},{Int32:7}])]
```

Secondly, we can construct the values in Opa and use `Bson.opa2doc`:

```
doc = Bson.opa2doc({`$eval`:(Bson.code "function(x,y) \{return x*y;}"),
                    args:(list(Bson.int32) [6,7])})
```

Notice that to get a field with non-alphanumeric characters we have to backquote the field name in the Opa value and that to control the representation in the BSON type we can apply helper types, for example `Bson.code` is just a string but it instructs `Bson.opa2doc` to treat it as code. Remember also to escape curly brackets in strings. Note that to get `Int32` values you need the `Bson.int32` type, the default for `int` is actually `Bson.int64`.

There are several such types provided by the `Bson` module but some merit special mention:

- Optional types have a special significance with respect to `Bson.doc2opa` in that if a field value is missing in the document it will appear in the Opa type as {`none`}. The alternate direction does not apply, {`none`} values are represented in the BSON document as { `none :  null` }.

```
type Bson.register('a) = {'a present} or {absent}
```

- We take this one step further, however, with the `Bson.register` type, which actually behaves much as `option('a)` except that when we call `Bson.doc2opa` any {`absent`} values are omitted from the resulting document altogether. Note that there is a module `Bson.Register` which provides the same functionality for `Bson.register` as the `Option` module does for type `option`.

- Care should be taken in dealing with integer values which may have been placed into the database outside of OPA. OPA uses, internally, the OCaml integer representation `int` which is actually 31 bits wide on 32-bit systems and 63 bits wide on 64-bit systems (the spare bit is reserved by the garbage collector). Now MongoDB actually uses fully 32-bit and 64-bit integers which means that it is possible to find an integer value in a MongoDB database which is too large for the OPA representation (remember that all values generated by OPA and stored in the database are guaranteed to be within range). Currently, OPA only has 32-bit and 64-bit integers as abstract values. Such values can be stored in OPA as an external type (`int32` and `int64`) but no operations are possible on these values (they are sometimes needed by external libraries). We handle this situation in the MongoDB driver by automatically detecting overflow values and storing them as `RealInt32` and `RealInt64` when returning `Bson.document` types from the driver. While these values may appear to be invisible to the `Bson` module functions such as `find_int`, you can detect overflows by inspecting the document values:

```
match (value) {
  case {RealInt32:_}: error("overflow");
  case {Int32:i}: i;
  default: error("not an int");
}
```

- The `Bson.meta` type is intended to support situations where MongoDB can return a field of different types depending upon the nature of the command executed. A good example of this is the `out` option to the `mapReduce` function which can be either a `string` or a document type. We cast the parameter as `Bson.meta` which allows us to control the type at the function's application. We can also apply this trick to the `result` type from `mapReduce` calls:

```
mr = MC.mapReduceSimple(mongodb,map,reduce,{String:"example1"})

/* or */

mr = MC.mapReduceSimple(mongodb,map,reduce,{Document:[H.str("reduce","session_stat")]})
```

- Two other cases should be mentioned. Both `list` and `intmap` are mapped onto `Array` values in BSON. The difference is that `list` is mapped to consecutive-numbered elements in the `Array` document whereas `intmap` allows sparse arrays.

As a rough guide to `Bson.opa2doc` and `Bson.doc2opa`, the following simple schema shows the mapping:

```
/* We use a "natural" mapping of constant types */
float <-> Double
string <-> String
Bson.binary <-> Binary
Bson.oid <-> ObjectID
bool <-> Boolean
Date.date <-> Date
void <-> Null
Bson.regexp <-> Regexp
Bson.code <-> Code
Bson.symbol <-> Symbol
Bson.codescope <-> CodeScope
Bson.int32 <-> Int32
Bson.realint32 <-> Int32
Bson.timestamp <-> Timestamp
Bson.realint64 <-> Int64
Bson.min <-> Min
Bson.max <-> Max

 /* Basic record scheme */
{a:'a; b:'b} <-> { a: 'a, b: 'b }

 /* Sum types */
{a:'a} / {b:'b} <-> { a: 'a } <or> { b: 'b }

 /* Non-record types are called "value" */
'a <-> { value: 'a }

 /* Special cases */

 /* Default for int is Int64 */
int <-> Int64

 /* Overflow */
Bson.realint32 <- Int32 /* when integer exceeds range */
Bson.realint64 <- Int64 /* when integer exceeds range */

 /* Options */
option('a):
  {some=a} <-> { some : 'a }
  {none} <-> { none : null }
  {none} <- { }

 /* Registers */
Bson.register('a):
  {present=a} <-> { present : 'a }
  {absent} <- { absent : null }
  {absent} <-> { }

 /* Lists are consecutive arrays */
list('a) <-> { Array=(<label>,{ 0:'a; 1:'a; ... }) }

 /* Intmaps are non-consecutive arrays */
ordered_map(int,'a) <or>
intmap('a) <-> { Array=(<label>,{ 1:'a; 3:'a; ... }) }

 /* Bson.document is treated verbatim (including labels) */
Bson.document <-> Bson.document

 /* Bson.meta is treated as a variable type */
int:Bson.meta <-> { Int64:int }
string:Bson.meta <-> { String:string }
bool:Bson.meta <-> { Boolean:bool }
etc.
```

Notes:

- For `ObjectID` values, there are a couple of routines which convert between (hex value) strings and the BSON representation, `Bson.oid_of_string` and `Bson.oid_to_string`. You can also create a BSON-style OID value with `Bson.new_oid`.
- `Bson.document` types are completely write-through, i.e. they are not processed at all.
- In case you're wondering, `Min` and `Max` are used in sharded databases to indicate infimum and supremum bounds on sharding regions, respectively.

//TODO: other functions find_xyz, to_pretty, error stuff

Using the low-level interface

Connecting to and using the low-level drivers should be done using the `MongoConnection` module. This gathers together various low-level features in a single module.

Opening a connection to the MongoDB server The preferred method is to use the system of named connections which can be defined from the command line or setup internally using the `Mongo.param` type and the `MongoConnection.add_named_connection` function.

Initially, there is one default connection (called ''default") which is set to `localhost:27017`, the default port for MongoDB servers on the local machine. To open this connection use:

```
mongodb =
  match (MongoConnection.open("default")) {
    case {success:mongodb}: mongodb
    case {~failure}: ... /* take action on error */
  }

/* or */

mongodb = MongoConnection.openfatal("default")
```

The `MongoConnection.open` function returns an outcome of either the connection or the standard `Mongo.failure` type whereas the `MongoConnection.openfatal` function returns just the connection but treats a failed connection as a fatal error.

To setup the connection from the command line the following options are defined:

{table} {* Option | Abbrev Type | Description *} {| `--mongo-name` | `(--mn)` <`string`> | Name for the MongoDB server connection |} {| `--mongo-repl-name` | `(--mr)` <`string`> | Replica set name for the MongoDB server |} {| `--mongo-buf-size` | `(--mb)` <`int`> | Hint for initial MongoDB connection buffer size |} {| `--mongo-socket-pool`| `(--mp)` <`int`> | Number of sockets in socket pool (>=2 enables socket pool) |} {| `--mongo-seed` | `(--ms)` <`host`>{`:`<`port`>} | Add a seed to a replica set, allows multiple seeds |} {| `--mongo-host` | `(--mh)` <`host`>{`:`<`port`>} | Host name of a MongoDB server, overwrites any previous hosts |} {| `--mongo-log` | `(--ml)` <`bool`> | Enable MongoLog logging |} {| `--mongo-log-type` | `(--mt)` <`string`> | Type of logging: stdout, stderr, logger, none |} {| `--mongo-auth` | `(--ma)` <`user:pwd@dbname`> | Define user name and password for database dbname |} {table}

So, for example, to connect to the default connection at `machinexyz:12345` you would use:

```
% prog.exe --mh machinexyz:12345
```

This remains a single connection, to connect to a replica set you also need to define a name for the replica set plus some seeds:

```
% prog.exe --mn blort --mr blort --ms machinexyz:27017 --ms machineuvw:27017
```

Here we have defined a connection called ‘’blort” to a replica set also called ‘’blort” with two seed machines. Remember that you only really need one seed which is active in the set, the connection logic queries the seeds for the actual host list and then polls the hosts until it finds the current primary server. From then on reconnection will be attempted if the current primary goes down.

Note that you can define as many named connections as you like, this example still retains the default connection.

Note also that you can clone a connection such that the connection itself will not be closed until all clones have already been closed.

Handling concurrency within an Opa program is done by a socket pool. This means that a pool of open connections is maintained to the same server such that blocking only occurs if there are no more available connections in the pool (set with `--mp 2`, for example). If you ensure that the pool size is at least as big as the number of threads in your code then no blocking will occur.

Named connections can also be defined within the program:

```
MongoConnection.add_named_connection({
  name: "blort",
  replname: {some: "blort"},
  bufsize: 50*1024,
  pool_max: 2,
  log: false,
  seeds:[("localhost",10001),("localhost",10002)],
  auth:[{dbname:"mydb",user:"me",password:"secret"}]
})

mongodb2 = N.openfatal("blort")
```

Once a connection has been opened, it can be pointed to different databases and collections using a functional interface. The default database is ‘’db” and the default collection is ‘’collection” but we can make a connection to a different collection without re-opening the connection as follows:

```
mongodb_wiki = MongoConnection.namespace(mongodb,"db","wiki")
```

This mechanism also applies to the flags that some of the MongoDB operations can take, for example to set the `Upsert` flag for all insert operations:

```
mongodb3 = MongoConnection.upsert(mongodb)
```

This method is quite flexible since you can define these flags once when the connection is made, making the flags globally persistent, or you can add these function calls at the point of calling the operation, i.e. locally defined flags (there are examples below). All of the MongoDB flags are supported in this way.

One particular flag is worth mentioning, the `log` flag which can be set on the command line and can actually be overridden in this way allowing you to generate logs for targeted sections of code. In fact, you can change any of the command line options this way but bear in mind that some of them, for example, seed lists, will not take effect until the connection is reconnected.

Authentication As you can see, you can add the MongoDB authentication parameters for a given database either on the command line using the `--mongo-auth` argument which is of the form: `user:password@database_name` or by placing the authentication parameters in the `auth` field in the `add_named_connection` function argument.

Alternatively, you can call the `MongoCommands.authenticate` function to perform an additional, external authentication. Note that if you are connecting to a replica set then the driver needs to re-authenticate after connecting to the new host so the authentication parameters are built into the low-level Mongo datatype. This means that if you call this function you should perform all subsequent operations on the returned Mongo datatype, not on the original which won't have the parameters built in.

Remember that authentication in MongoDB is to a database, not to a connection so you can have multiple user names and passwords associated with a single connection. If you want to authenticate with all of the databases over a connection you need to authenticate with the `admin` database which acts a bit like ''root" access for databases.

Basic operations The basic database access operations are the same as the MongoDB protocol operations, i.e. insert, update, query, get_more, delete, kill_cursors and msg. So, for example, to insert a document:

```
/* A couple of documents */
p1 = [H.str("name","Joe1"), H.i32("age",44)]
p2 = [H.str("name","Joe2"), H.i32("age",55)]

/* Insert the documents */
MongoConnection.insert(mongodb,p1)
MongoConnection.insert_batch(mongodb,[p1,p2])
```

The basic write operations come in three types:

- `insert` is the write-and-forget operation where the insert message is sent and a boolean value is returned which simply states that the correct number of bytes were written to the socket.

- `inserte` is a ''safe" operation where the insert message has a `getlasterror` query piggy-backed onto it and then the raw optional reply is returned.
- `insert_result` does an `inserte` and then analyzes the reply, turning it into a standard `Mongo.result` type.

All of the basic write operations have these three forms. The `Mongo.result` type is an `outcome` of either success as a `Bson.document` type or failure as a `Mongo.failure` type. The `Mongo.failure` type looks like:

```
type Mongo.failure =
    {OK}
 or {string Error}
 or {Bson.document DocError}
 or {Incomplete}
 or {NotFound}
```

This defines either a raw document error {`DocError:doc`} which is an error as reported by the MongoDB server, a driver error {`Error:str`} which is a message generated by the Opa driver or a few special-purpose errors returned under specific circumstances ({`OK`} is simply a connection that has never been used).

Post-processing of results may include checking for errors:

```
error = MongoConnection.insert_result(MongoConnection.upsert(mongodb),[H.i32("i",n)])
println("insert error={MongoCommon.is_error(error)}")
```

or extracting specific fields from the reply:

```
println("errmsg={MongoCommon.result_string(error,"errmsg")}")
```

noting that we also support the MongoDB dot notation syntax:

```
println("indexSizes._id_={MongoCommon.dotresult_int(collStats,"indexSizes._id_")}")
```

Closing a connection is as simple as:

```
MongoConnection.close(mongodb)
```

Remember that the connection will only close once all of the clones have also been closed.

Cursors Handling queries in MongoDB has the complication that, for efficiency, cursors are stored on the server which entails tracking them at the client side. While the bare `MongoConnection.query` and `MongoConnection.get_more` operations can be used to handle queries in conjunction with the reply support code in `MongoCommon` they are a bit inconvenient.

For this purpose we have defined cursor operations in the `MongoCursor` module and re-exported the most important ones into the `MongoConnection.Cursor` module. A cursor object itself contains all the parameters needed to manage the cursor at the server side and, in fact, duplicates some of the information in the connection object. Using the re-exported functions reduces the number of parameters to the basic functions since this information can be lifted from the connection into the cursor object.

Here is an example of a low-level cursor dialog:

```
cursor = MongoConnection.Cursor.init(mongodb)
cursor = MongoConnection.Cursor.set_query(cursor,{some:[H.str("name","Joe")]})
cursor = MongoConnection.Cursor.set_limit(cursor,3)
cursor = MongoConnection.Cursor.set_fields(cursor,{some:[H.i32("_id",0)]})
cursor = MongoConnection.Cursor.next(cursor)
result = MongoConnection.Cursor.check_cursor_error(cursor)
println("result 1 = {MongoCommon.pretty_of_result(result)}")
println("valid 1 ={MongoConnection.Cursor.valid(cursor)}")
cursor = MongoConnection.Cursor.next(cursor)
result = MongoConnection.Cursor.check_cursor_error(cursor)
println("result 2 = {MongoCommon.pretty_of_result(result)}")
println("valid 2 = {MongoConnection.Cursor.valid(cursor)}")
MongoConnection.Cursor.reset(cursor)
```

The cursor is initialized with `init` and then the parameters for the query are setup. The `next` function generates the `query` (or `get_more`) call to the server and places the next document internally in the cursor object along with any error status. The `check_cursor_error` function is a convenient way of extracting either the current document or the error as a `Mongo.result`. Subsequent calls to `next` will either return the next document from the previous reply or issue a `get_more` call to re-populate the cursor. The end of the matching documents (or if no document matches) is signaled with `NotFound` and if you try to read past the end of matching documents you will get an ''end of data" error from the driver. The `valid` function is used to poll whether there is any remaining data. Finally, the call to `reset` is important here because it doesn't just end the query, it will issue a `kill_cursors` operation to the server to tell it to delete the cursor (cursors time out after 10 minutes by default on the MongoDB server).

This method works fine but this logic has been wrapped up into some convenience functions:

- `find_one` returns the first matching document as a `Mongo.result`
- `find_all` gives all the matches as a list of documents (use the `limit` function to limit the number of replies).

For example:

```
/* Find all objects in db.session, excluding the _id field */
mongo_session_no_id =
  MongoConnection.fields(MongoConnection.namespace(mongodb,"db","session"),{some:[H.i32("_ic
println("findAll: {CM.pretty_of_results(MongoConnection.Cursor.find_all(mongo_session_no_id,
```

You can also define custom loops over the matches using `start` (or `find`) in conjunction with `next` and `valid`. (Note that you must use the `MongoConnection.Cursor.for` loop instead of the more usual `for` function in the Opa stdlib, you need to check for valid and only call next if still valid at that point, otherwise you will miss the last document in the list of matches).

//Commands //~~~~~~~~~

Collections

While you can achieve anything that MongoDB is capable of using the low-level drivers, there are no guarantees of type safety while converting between BSON documents and Opa values. You can of course base your entire project around BSON values and eliminate the need for converting between MongoDB's documents and Opa types altogether but this may not be very convenient depending upon what is happening elsewhere in your application. Secondly, to use the low-level drivers requires an investment in learning MongoDB's powerful but rather complex interface (which may be new to users of relational databases) in order to exploit what MongoDB has to offer. Finally, basing your application on MongoDB's API will tie your application to MongoDB and you may at some point in the future wish to migrate to other database solutions.

Ultimately, the intention is to provide an abstract view of the database which is general enough to encompass several of the existing database solutions, of which MongoDB is an important player, and support this with compiler-generated syntax in the manner of the Opa inbuilt database. This support is still not available but we can offer an intermediate layer of programming MongoDB whereby we assume collections of Opa types and support type-safety by performing run-time type-checks on operations over these collections. This support is in the form of the `MongoCollection` module plus some support modules for generating values suitable to be applied to these functions.

The `collection` type

The central idea in the `MongoCollection` module is a collection (in the MongoDB terminology sense) of Opa values. This is embodied in the `Mongo.collection` type which is extremely simple, it's just a `MongoConnection` value cast to the specific type of the values to be stored in the collection:

```
type Mongo.collection('a) = {
  Mongo.mongodb db /* the mongodb connection */
}
```

When a value is stored in the collection it is automatically converted from its Opa type into a matching BSON document and *vice versa* for queries.

While this sounds simple there are a number of pitfalls to watch out for. We assume that any offline modifications of the collection will not create any incompatible values. If, for example, we add or delete a field from a record then the entry can no longer be represented as an Opa type.

To overcome this problem we place checks in the code to verify the suitability of documents read from the collection and an error will be generated if any such values are found. We also provide features to allow handling of this situation in some specific circumstances, for example, if you type a field in the collection as `Bson.register` it will allow you to successfully read in values with missing fields but this is not recommended for collections. Ultimately, it is up to the maintainer of the database to ensure that the values stored there are consistent with the application's usage of the collection.

Despite these provisos, using a collection is very simple and gives the programmer the ability to integrate Opa types with the MongoDB system without having to understand the underlying complexity of the database and with a modest level of type-safety. The cost, for the moment, is the overhead of the run-time type-checks which will slow down database operations.

Programming with collections

A simple dialog for creating and manipulating a collection might be as follows:

```
/* The type of our first collection */
type t = {int i}

/* Create a collection of type t */
Mongo.collection(t) c1 = MongoCollection.openfatal("default","db","collection")

/* Put a single value into the collection */
result = MongoCollection.insert_result(c1,{i:0})
```

```
/* Finally, destroy the collection */
MongoCollection.destroy(c1)
```

We define a type for the collection (`type t`) so that when we open a connection to the database we can cast the resulting collection object and thus install the correct run-time representation of the type. The `openfatal` function returns a collection and treats a connection failure as fatal. There are several variants of the `open` function.

A collection is a pointer to a specific collection in the database (here, `db.collection`) and we create a connection to the MongoDB server using the connection name (in this instance, `default`).

Inserting a value into the collection is trivial, the value is simply passed as it is to the `insert` function (here we use the safe `insert_result` function which also returns the result of a `getlasterror` call). The insert has exactly the same effect as a call to `MongoConnection.insert` but with the value automatically converted into a BSON document using the scheme outlined above.

The call to `MongoCollection.destroy` should not be forgotten because this closes the underlying connection.

While the `insert` function is trivial, we need more care with `update` and `delete`. The problem is that to maintain our level of type-safety we need to match select (and update) documents with the type of the collection they are applied to. We do this with a system of run-time type-checks applied to the select documents. For example:

```
/* Create pre-typed select and update generation functions */c
MongoSelect.create reatest = Bson.document -> Mongo.select(t)
MongoUpdate.create createut = Bson.document -> Mongo.update(t)

/* Generate the select documents */
select = createst(MongoSelectUpdate.int64(MongoSelectUpdate.empty(),"i",0))
update = createut(MongoSelectUpdate.inc(MongoSelectUpdate.int64(MongoSelectUpdate.empty(),"i

/* We can now apply update to these documents */
result = MongoCollection.update_result(c1,select,update)
```

Firstly, we use the `MongoSelectUpdate` module to generate the basic documents. Note that we could also have used the `Bson.opa2doc` function to achieve the same result:

```
select = createst(Bson.opa2doc({i:0}))
update = createut(Bson.opa2doc({‘$inc‘:{i:1}}))
```

The choice between these two styles may depend upon the type of document being generated. The Opa type-based versions are more readable but the `MongoSelectUpdate` ones are much faster since no conversion is required.

The select documents have to be correctly typed for the collection they apply to so we generate a couple of convenience functions `createst` and `createut` to do the casting for us.

Secondly, once we have these documents we can apply the `update` function to them but note that although a select document is just a typed `Bson.document` it triggers a set of suitability tests. These tests are complex and probably do not cover all possible MongoDB operations but briefly, the select document is scanned by a knowledge-base of the types of MongoDB field types, for example `$inc` only applies to updates, `$and` only applies to selects whereas `$comment` can apply to both. Once the status (select/update/both) is determined, the type of the resulting values is determined from the select document and is verified to be a subtype of the type of the collection. So, for example, {`int a`} is a subtype of {`int a, string b`} but {`int a, bool c`} is not. Presently, we only print a suitable warning but in future, once these routines have fully matured we may return an error value.

All of the basic database write operations occur in both send-and-forget and in send-with-getlasterror forms: `insert`, `insert_result`, `insert_batch`, `insert_batch_result`, `update`, `update_result`, `delete` and `delete_result`.

As an aside, notice that we use a similar functional interface for flags as for the low-level code:

```
MongoCollection.delete(MongoCollection.singleRemove(c1),createst(Bson.opa2doc({i:104})))
```

The select mechanism applies to queries as well but in this case we have to be careful what types we return from the database:

```
result = MongoCollection.find_one(c1,createst(Bson.op12doc({‘$where‘:(Bson.code "this.i > 1
match (result) {
  case {success:{~i}}: println("i={i}")
  case {~failure}: println("error={MongoCommon.string_of_failure(failure)}")
}
```

This example returns the first value in the collection for which `i` is greater than 106, it expresses the select as a Javascript expression. Many of the MongoDB query methods are perfectly safe with collections such as the `$where` example here but some methods are not safe in that they return documents which contain fields other than those in the Opa type, a good example being the http://www.mongodb.org/display/DOCS/Explain[`$explain`] documents which are a set of statistical data concerning the given query (see the `Mongo.explainType` type in `MongoCommands`). In general, we attempt to support such features with special purpose functions rather than via the normal database operations.

The usual simplified query functions are present in `MongoCollection`, `find_one` and `find_all`. There are also two functions which return the bare `Bson.document` representation of the result, `find_one_doc` and `find_all_doc` which may be useful in the above situation where the result of the query is not compatible with Opa types. For more general query scanning, the cursor-based routines are available. For example, the following code scans the results of a `MongoCollection` query

```
query = createst(Bson.opa2doc({i:{‘$gt‘:102, ‘$lt‘:106}}))
match (MongoCollection.query(MongoCollection.limit(c1,0),query)) {
  case {success:cc1}:
    cc1 =
      while(cc1,(function(cc1) {
                   match (MongoCollection.next(cc1)) {
                     case (cc1,{success={~i}}):
                        println("i={v}")
                        (cc1,MongoCollection.has_more(cc1))
                     case (cc1,{~failure}):
                        println("error={MongoCommon.string_of_failure(failure)}")
                        (cc1,false))})
    MongoCollection.kill(cc1)
  case {~failure}:
    println("error={MongoCommon.string_of_failure(failure)}")
}
```

In this code, we create a `Mongo.collection_cursor` object using `MongoCollection.query` to which we can then apply the collection-specific cursor functions `MongoCollection.next` and `MongoCollection.has_more`. This allows arbitrary processing of collection queries. Remember, as with the low-level cursors above, that the `MongoCollection.kill` function does not just end the scan, it also sends a `kill_cursors` message to the MongoDB server to tell it to destroy the cursor.

Another aside in this code is that we set the `limit` value to `0` which means ‘’use the default number of documents per reply”. If we had set this to `1` we would only ever get one document in the reply because MongoDB treats this as a special case, i.e. ‘’just return one document”.

Again, to help with the situation where return values may be incompatible with Opa types, we provide the `_unsafe` variants of the query functions. These, for example `query_unsafe`, take an additional boolean flag, `ignore_incomplete` which instructs the driver to simply ignore any return documents which have missing fields and are thus not compatible with Opa types. MongoDB will actually return partial documents if the document meets the query document but does not contain all of the fields (an exception is the _id field which is always returned unless specifically excluded with the return field selector document). These functions should be used with care.

Apart from the support described here the MongoCollection module also provides a few convenience functions such as creating indexes using collection objects and some direct support for some of the aggregation functions (count, distinct and group). Finally, one of the variants of the open function, openpkg and openpkgfatal supplies a set of pre-cast versions of MongoSelect.create and MongoUpdate.create.

Example: Hello, MongoDB wiki

In this section, we describe how to convert the hello_wiki example described in the previous chapter to using the MongoDB database. This is actually a simple process and uses MongoDB as a simple key-value storage database.

// TODO: more realistic example

The first task is to open a connection to the database. We are going to use collections and in fact, we will use the version of open which also gives us the casting functions for selects:

```
/**
 * The basic info. about the database and table location.
 */
type page = {
  string _id,
  Bson.int32 _rev,
  string content
}

/**
 * We work at level 1, run-time type-checked storage of a collection of Opa values.
 * The Mongo.pkg type provides convenience functions for building select and update document
 **/
Mongo.pkg(page) (wiki_collection,wiki_pkg) = MongoCollection.openpkgfatal("default","db","wi
function pageselect(v) { wiki_pkg.select(Bson.opa2doc(v)); }
function pageupdate(v) { wiki_pkg.update(Bson.opa2doc(v)); }
```

The _rev field has been cast to Bson.int32 so we can use 32-bit integers for this field (it is unlikely we will ever have more than 4 giga-revisions of any value in the database!). We then open our connection using the default named connection and connect to the collection db.wiki. This returns a collection object plus a package of values which we use to build our select documents.

Next we are actually going to search for documents including the `_rev` field so we can't just use the default index for our collection (the `_id` field):

```
/**
 * Indexes aren't automatic in MongoDB apart from the non-removable _id index.
 * Since we're searching on _rev as well, we need a separate index.
 **/
MongoCollection.create_index(wiki_collection, "db.wiki", Bson.opa2doc({_id:1; _rev:1}), 0)
```

The `get_content` function can then be modified using a simple call to `MongoCollection.find_one`:

```
function get_content(docid) {
  default_page = "This page is empty. Double-click to edit."
  function extract_content(page record) { record.content }
  /* Order by reverse _rev to get highest numbered _rev. */
  orderby = {some:Bson.opa2doc({_rev:-1})}
  match (MongoCollection.find_one(MongoCollection.orderby(wiki_collection,orderby),pageselec
    case {success:page}: extract_content(page)
    case {failure:{NotFound}}: default_page
    case {~failure}:
      jlog("hello_wiki_mongo: failure={MongoCommon.string_of_failure(failure)}")
      default_page
  }
}
```

We search the database for the given `_id` value but we want the highest-numbered `_rev` field so we sort by inverse order on that field (the default ordering for numerical fields is in increasing order). A missing document is signaled by the `NotFound` failure condition, other `failure` values are errors.

Finally, the `save_source` function becomes a call to `MongoCollection.update_result`:

```
exposed function save_source(topic, source) {
  select = pageselect({_id:topic})
  update = pageupdate({`$set`:{content:source}, `$inc`:{_rev:(Bson.int32 1)}})
  /* Upsert this so we create it if it isn't there */
  result = MongoCollection.update_result(MongoCollection.upsert(wiki_collection),select,upda
  if MongoCommon.is_error(result)
    then <>Error: {MongoCommon.pretty_of_result(result)}</>;
    else load_rendered(topic);
}
```

In this case, we select only the `_id` field and we update the document by setting the `content` field and incrementing the `_rev` field. Note that we use the `Upsert` flag which tells MongoDB to insert the document if it isn't already present in the collection. We test the result for errors using the safe update operation but apart from that the code is identical to the existing ''Hello wiki" example.

Running Executables

// // About this chapter: // Main author: ? // Paired author:? // // Topics: // - The compiler // - Launching and configuring applications // - Launching and configuring distributed stuff // - External tools (e.g. db recovery, db tool, xml import/export, opadoc) //

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.

Level and Modes of use

The compiler offers 2 levels of use, a high-level one, and a low-level one, and 2 low-level modes of execution, the compilation and the linking.

Standard usage The high-level usage 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
```

Other modes The low-level usage of *opa* is comparable with standard compilers like *gcc* or *java*. It takes one or 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.

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.

Linking The linking is the final step in the build process. This produces an standalone executable from given previously compiled *opx* files.

{block}[TIP] ##### 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*) {block}

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
```

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]

{table} {* 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 |} {table}

Output of the compiler

The compiler produces a number of files and/or executable, depending on its mode of usage, and the options activated.

{table} {* output | mode or option | description *} {| api | –api | used by *opadoc* |} {| executable | linking or autobuild | standalone executables |} {| odep | –odep | dependency graphs of the compiled application |} {| opx | compilation or autobuild | compiled packages |} {table}

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*

Levels and modes of usage {table} {* option | argument | description *} {| | 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 libraries |} {table}

// Other options // +++++++++++++

// TODO!

// Common errors // ^^^^^^^^^^^^^

// TODO!

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:

{table} {* 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 |} {table}

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*

{block}[TIP] ##### 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) {block}

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'
```

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.)

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

{block}[CAUTION] *api* files are not given in the command line of *opadoc*, but found automatically near *opa* files or in the given directories {block}

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

html A directory is generated containing several html files, with a list of types, values, packages, etc.

Options

{table} {* option | argument | description *} {| –output-dir, -o | | 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. |} {table}

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
```

// Common errors // ^^^^^^^^^^^^^

// TODO!

// Syntax of comments // ^^^^^^^^^^^^^^^^^^

// TODO!

// opa-plugin-builder TODO!

// Coming soon...

////

//Arguments, input //Output //Options //Common errors //Warnings //[|opa_plugin_browser|] //opa-plugin-browser //Coming soon... //Arguments, input //Output //Options //Common errors //Warnings ////

Opa applications, at run-time

This section details the use of applications built with Opa, including:

- command-line arguments;
- logging of events;
- network administration;
- environment variables.

// // Main editor for this section: Louis Gesbert //

/// // If an item spans several sections, please provide // hyperlinks, e.g. type definitions have both a syntax // and a more complete definition on the corresponding // section ///

/// // If an item is considered experimental and may or may // not survive to future versions, please label it using // an Admonition block with style [CAUTION] ///

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).

Debugging resources

[[runtime_editable_resources]]

Opa applications 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.

//### Index of command-line arguments //- //- //- //- //-

Running Executables

// // About this chapter: // Main author: ? // Paired author:? // // Topics: // - The compiler // - Launching and configuring applications // - Launching and configuring distributed stuff // - External tools (e.g. db recovery, db tool, xml import/export, opadoc) //

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.

Level and Modes of use

The compiler offers 2 levels of use, a high-level one, and a low-level one, and 2 low-level modes of execution, the compilation and the linking.

Standard usage The high-level usage 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
```

Other modes The low-level usage of *opa* is comparable with standard compilers like *gcc* or *java*. It takes one or 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.

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.

Linking The linking is the final step in the build process. This produces an standalone executable from given previously compiled *opx* files.

{block}[TIP] ##### 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*) {block}

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
```

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]

{table} {* 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 |} {table}

Output of the compiler

The compiler produces a number of files and/or executable, depending on its mode of usage, and the options activated.

{table} {* output | mode or option | description *} {| api | –api | used by *opadoc* |} {| executable | linking or autobuild | standalone executables |} {| odep | –odep | dependency graphs of the compiled application |} {| opx | compilation or autobuild | compiled packages |} {table}

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*

Levels and modes of usage {table} {* option | argument | description *} {| | 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 libraries |} {table}

// Other options // +++++++++++++

// TODO!

// Common errors // ^^^^^^^^^^^^^

// TODO!

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:

{table} {* 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 |} {table}

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*

{block}[TIP] ##### 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) {block}

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'
```

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.)

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

{block}[CAUTION] *api* files are not given in the command line of *opadoc*, but found automatically near *opa* files or in the given directories {block}

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

html A directory is generated containing several html files, with a list of types, values, packages, etc.

Options

{table} {* option | argument | description *} {| –output-dir, -o | | 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. |} {table}

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
```

// Common errors // ^^^^^^^^^^^^^

// TODO!

// Syntax of comments // ^^^^^^^^^^^^^^^^^^

// TODO!

// opa-plugin-builder TODO!

// Coming soon...

////

//Arguments, input //Output //Options //Common errors //Warnings //[|opa_plugin_browser|] //opa-plugin-browser //Coming soon... //Arguments, input //Output //Options //Common errors //Warnings ////

Opa applications, at run-time

This section details the use of applications built with Opa, including:

- command-line arguments;
- logging of events;
- network administration;
- environment variables.

/// // Main editor for this section: Louis Gesbert ///

/// // If an item spans several sections, please provide // hyperlinks, e.g. type definitions have both a syntax // and a more complete definition on the corresponding // section ///

/// // If an item is considered experimental and may or may // not survive to future versions, please label it using // an Admonition block with style [CAUTION] ///

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).

Debugging resources

[[runtime_editable_resources]]

Opa applications 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.

//### Index of command-line arguments //- //- //- //- //-

//[appendix] 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

// TODO: add hyperlink to other part mentionning and/or handling these files
// exemple: api ==> opadoc, etc.

// Alphabetically

api

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.

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
```

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
```

// TODO hyperlink=>opadoc

api-txt

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).

Produced by

Same than *api* files, generated by *opa* with the option `--api`

Used by

Meant to be read by an human, e.g. for debugging.

conf

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.

Produced by

Hand written by an author of an opa application.

Input

opa

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
```

{block}[TIP] ### 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 {block}

{block}[TIP] ### About package_name

They follows the same conventions as the *import* construct in Opa, but no spaces are allowed using the extension syntax, as in these examples: examples:

```
import stdlib.*
import toto.{foo,bar}
```

{block}

//[float] ##### 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
```

jsconf

Description

conf for bsl js files, Cf part about plugin for the syntax of these files

Produced by

Hand written by the author of a js plugin.

Used by

opa-plugin-builder

opa

Description

This is the extension of the Opa files. Most of the files in an Opa project are .opa files

Produced by

Hand written by authors of an Opa application.

Used by

opa, *opadoc*

opack

Description

opack files are used to group the command line options and arguments for invoking *opa* to build an Opa application.

Example:

```
user@computer:~/$ opa myproject.opack
```

is almost equivalent to

```
user@computer:~/$ cat myproject.opack | grep -v '#' | xargs opa
```

Produced by

Hand written by authors of an Opa application.

Used by

opa

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

Example

```
# This is an example of opack file
myfile_1.opa
myfile_2.opa
--warn-error root
```

//[[filenames_opp]] opp —

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*.

Produced by

opa-plugin-builder

Used by

opa, *opa-plugin-browser*

opx

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.

Produced by

opa

Used by

opa

//[appendix] 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

// TODO: add hyperlink to other part mentionning and/or handling these files
// exemple: api ==> opadoc, etc.

// Alphabetically

api

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.

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
```

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
```

// TODO hyperlink=>opadoc

api-txt

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).

Produced by

Same than *api* files, generated by *opa* with the option `--api`

Used by

Meant to be read by an human, e.g. for debugging.

conf

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.

Produced by

Hand written by an author of an opa application.

Input

opa

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
```

{block}[TIP] ### 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 {block}

{block}[TIP] ### About package_name

They follows the same conventions as the *import* construct in Opa, but no spaces are allowed using the extension syntax, as in these examples: examples:

```
import stdlib.*
import toto.{foo,bar}
```

{block}

//[float] ##### 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
```

jsconf

Description

conf for bsl js files, Cf part about plugin for the syntax of these files

Produced by

Hand written by the author of a js plugin.

Used by

opa-plugin-builder

opa

Description

This is the extension of the Opa files. Most of the files in an Opa project are .opa files

Produced by

Hand written by authors of an Opa application.

Used by

opa, *opadoc*

opack

Description

opack files are used to group the command line options and arguments for invoking *opa* to build an Opa application.

Example:

```
user@computer:~/$ opa myproject.opack
```

is almost equivalent to

```
user@computer:~/$ cat myproject.opack | grep -v '#' | xargs opa
```

Produced by

Hand written by authors of an Opa application.

Used by

opa

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

Example

```
# This is an example of opack file
myfile_1.opa
myfile_2.opa
--warn-error root
```

//[[filenames_opp]] opp —

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*.

Produced by

opa-plugin-builder

Used by

opa, *opa-plugin-browser*

opx

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.

Produced by

opa

Used by

opa

Opa is an open source, web development platform designed specifically for the Web with both security and developer agility in mind.

Opa provides a complete stack for web application development based on a new programming language integrated with its web server, database engine and distribution libraries. This unique setup and tight coupling allows correctness, productivity and security.

Due to the strong static typing of the Opa language, a large spectrum of programming errors will be detected by the compiler without having to explicitly write any type annotations.