`b2rust` specification

A C++ B-to-Rust converter

Elouan Fabre, Paris.
June 26, 2023.

1 Preamble

1.1 Abstract

`b2rust` is written in C++ and converts a B program into a Rust library (a set of Rust code files) which can be compiled without errors (but not necessarily without warnings). The functions it defines can be used in a separate Rust project. Its calling syntax (section 4.1) is the following:

```
b2rust module -I directory
   [-O output_directory]
```

If an `output_directory` is not provided, the Rust library will be generated in the current directory.

Input requirements (section 4.1):

- `module` is the base module or entry module: the module which recursively imports every module and which is not imported by any module.
- `directory` must be accessible and contain the base module (so, the files `directory/module.bxml` and `directory/module_i.bxml` must exist and be readable) as well as any imported component (which have to be implemented). All these files have to be BXML 1.0 compliant files.
- `b2rust` needs to have write access to `output_directory`. It should not already contain files whose name are under the shape `i.rs`, where `i` is the name of a module in `directory`.

If these requirements are not respected, an unknown behaviour could happen.

If the B module associated with the `module.bxml` and `module_i.bxml` as well as recursively imported modules codes match some specifications (section 5), `b2rust` shall generate a Rust library (a code file for each module whose name is the name of the module) which matches the specification of the B entry module. To put in a nutshell, `b2rust` translates variables, constants, assignments, `if`/`then`/`else`, asserts, local variables definitions, operation definition with any number of input or output parameters, imports, operation calls, some expressions using tabulars, integers, booleans, basic maths.

The library can be compiled with the command `rustc --crate-type=lib a.rs` (if `a.rs` is the generated file). The library can then be used in a Rust project (with a `main`), using the compilation option `--extern out=libout.rlib` (section 4.3).

Run `cmake . && make` if you want to compile, `ctest .` if you want to run the tests (section 3.3).

1.2 About this document

This document introduces a B-to-Rust converter written by Elouan Fabre in CLEARSY in 2022 and 2023. It should be the entry point for anyone who:

- Just wants to compile `b2rust` and run the test bench;
- Seeks to use `b2rust` on a B0 project written by Atelier B, given its BXML associated files;
- Wants to understand the functionalities of `b2rust` or its conversion scheme;
- Wants to improve `b2rust`.

Every remark you have can be sent by mail to the following address: `elouan.fabre@clearsy.com` at least until June 27$^{\text{th}}$, 2023. After this date, you should send a mail to CLEARSY. Anyway, I am, until this same date, open to pull requests correcting this document (especially concerning the spelling or the grammar corrections).

1.3 Table of contents

2 Introduction

The *B method* is a good way to write formally verified programs when dealing with highly sensitive field of activity like nuclear activities or railway. It consist in writing a program specification using the B language, and then progressively refining it to a program written in a Turing-complete subset of the B language[1], for example, B0. A B0 program is quite close to a program written in a high-level language like Python or BASIC and is fairly algorithmic. However, unlike most programming languages, B0 has no default dedicated compiler to machine

[1] With some exceptions; it is the case here with `b2rust` which can translate a "non-Turing B instruction" (in B terminology, a B substitution which is not a B instruction) which is the $\text{Substitution\_devient\_elt\_de}_{\text{b2r}}$ substitution.

code. It can seem disconcerting because formally verified machine code is, undoubtedly, the final goal of formal methods. Some still exist, like B32, which compiles the B0 code to MIPS assembly. However, due to safety concerns, it is much preferable to write several converters to different programming languages and then compare their results on a same B0 program.

These last years, the Rust language has gained some attractiveness in the field of the safety. Rust has some nice features to provide an error-free program, for example by preventing the definition of references to freed objects. Some good converters exist like ComenC for the C language, but there were, until today, no converters for the Rust language. This is the reason why CLEARSY has suggested the creation of a B-to-Rust converter.

However, `b2rust` won't recognize exactly B0 which is an arbitrary subset of the B language; but, as B0 quite stands as a reference in the B environment, it shall be referenced a lot through this document.

Depending on the goal of the reader, a section or another can raise your interest. You may find below the different sections of this document, their purpose and their reference:

- How to compile and test `b2rust` (section 3);
- How to use `b2rust` (section 4);
- Which types of B0 components does `b2rust` correctly parse and translate (section 5);
- What is the translation specifications of `b2rust` (section 6);
- How does `b2rust` work (section 7);
- What are the development conventions of the development of `b2rust` (section 8).

3 Setup

3.1 Compilation of `b2rust`

The compilation needs CMake, Make, and a C++ compiler. Manage to have them installed on your computer. `b2rust` needs `tinyxml2`, a library used to read the BXML file (which is a specific XML file), and already used in Atelier B. If you want to compile `b2rust`, you need to have the headers and the dynamic libraries of `tinyxml2` installed. On Void Linux, this is done with the following command:

```
# xbps-install tinyxml2-devel tinyxml2
```

Now, get the `b2rust` source code and navigate to the file. On at least all UNIX platforms, you probably just need the following commands:

```
$ cmake .
$ make .
```

3.2 Generate the documentation

For now, the documentation only consists of the document you are reading (`b2rust` specification), which is written with LuaLaTeX, successor of LaTeX. The PDF file is in the `doc` folder (it's `specification.pdf`), however, if you update the source file `specification.tex` which you can find in the same directory, or if you just want a document freshly compiled, you can just navigate to the directory and compile the document. To do this, you need a TeX distribution which provides LuaLaTeX and the packages used in the `.tex` document. On Void Linux, you can achieve this with the following command:

```
# xbps-install texlive-full
```

However, this is a little "overkill" and you probably can install a bit less than all the LaTeX packages, thus ending up with an installation of less than several gibibytes. Anyhow, you can generate the PDF by navigating to the `doc` folder and using the command:

```
$ lualatex --shell-escape specification.tex
```

(The `--shell-escape` flag is needed by the `minted` package which allows to print code.)

3.3 Testing

`b2rust` comes with an adaptative test bench. It is configured with CMake, but before executing `ctest .` to run all the tests, you need to run `cmake .` first, for a reason explained later. A test consists of a directory of a certain name. There are four different types of tests: *call_error* tests, *comparison* tests, *error* tests and *main* tests. A test of name n is the directory `tests/`c`/`n, where c is the name of its category. For any category c, a `tests/`c`/test.sh` file exists; its role is to proceed with a certain test of the c category. If the test requires BXML files, `b2rust` takes as input the directory `tests/`c`/`n and n as name of the module to use as entry point.

```
[do_test.sh] Proceeding test `skip_operation`... Info: you can run yourself the
code generation part of the test with the command /home/efabre/Documents/b2rust/bin/b2rust
-a /home/efabre/Documents/b2rust/tests/bxml//skip_operation/skip_operation.bxml
-i /home/efabre/Documents/b2rust/tests/bxml//skip_operation/skip_operation_i.bxml.
 Test failed.
The reference file is

 fn main() {
 }
 fn skip_m() {
 }

but the file generated by b2rust is:

 fn main() {
 }
 fn skip_me() {
 }

If you don't see the difference, here's a `diff` output:

 3c3
 < fn skip_me() {
 ---
 > fn skip_m() {
```

Figure 1: Example of output of `do_test.sh` in case of failed test. This output is the one of an old version of `b2rust`, so, don't be surprised by this output.

1. A *call_error* calls `b2rust` with calling options which should fail. It is used to test the parameters parsing. If the test name is n, the file `tests/`c`/`n`/command` contains the parameters to append to the call of `b2rust` and `tests/`c`/`n`/error` contains the error code we should get. If the programs fails with a bad error code, the test fails. Otherwise, it succeeds.
2. A *comparison* test compares the output file `b2rust` generates for the entry module with a `.rs` file. If these two files are 100 % identical, the test succeeds. Otherwise, it fails. If a test fails, a `diff` output is printed. It can be quite useful (figure 1). In any case, the output of `b2rust` is printed, as well as the executed command. If the test name (i.e. the directory name) is n, the output of `b2rust` is compared with the `r_`n`.rs` file.
3. A *error* test is similar to a `call_error` test, but the changing parameter is the content of the BXML files (and not the calling parameters). It is

useful because `b2rust` shan't output any code if the input files do not respect the specifications. However, it should not throw a segmentation fault error, for example.

4. A *main* test is a more interesting test. It does the following:
   (a) It executes `b2rust` with the given directory and module name;
   (b) The code associated to the entry module and produced by `b2rust` is compiled to a library;
   (c) A Rust code file provided by the user is also compiled with linking to the created library;
   (d) The created binary is executed.

   If all these steps succeed (i.e. with the error code 0), the test succeeds. Otherwise, it fails. If the name of the test is n, then, the code provided by the user needs to be `tests/main/`n`/main.rs`. This test is interesting because it simulate the action an end-user would do like specified in the section 4.3.

The test bench is *adaptative*: it is generated each time you run `cmake ..`

Its inner working is not complicated; for each category c of tests, it goes over each directory of `tests/`c, and generate a line inside the `tests_file.cmake` file (the one CMake executes when you run `cmake .`) telling it to run the c`/test.sh` script with the correct arguments matching to the one we want for the found test. Note that the generation of tests is "blind" as it doesn't check if the tests match the specification of their category given above.

Notice the script `gen_bxml.sh` script; please do not consider it as a part of the `b2rust` environment, but rather as a script you might find helpful. It can help you adding your own tests from `.mch` and `.imp` files rather than from `.bxml` files. To do this, you just need to provide the Atelier B `bbin` directory which contains its executables as well as the path to the `tests` directory and Atelier B's ressource file. The script shall go over all the tests of every category (i.e. all the tests) and generate every BXML file it can (it goes over all the found `.mch` and `.imp` files). It is the reason why you may found `.mch` and `.imp` files in each BXML reference folder; they are just "convenience" files which do not match any specification, but you may find them helpful. `gen_bxml.sh` has hardcoded paths to BXML files it shouldn't generate; it is used for `errors` tests.

You might find another script useful: `test_command.sh` which just starts `b2rust` on a given test (so that you doesn't have to type its arguments which can be long in the case of a test).

In any case, some useful information are printed during the tests, but CMake doesn't print the output of the tests by default. If you want to print them, run `ctest . -V`. Note that `ctest . -V -R test_3615` will print the output of `do_test.sh` for the test `test_3615`.

4 Using b2rust

This section shall explain how one can use `b2rust` and the calling specification he would have to respect but without going into too much detail; in particular, it doesn't explain the specifications the BXML files have to respect. It is the role of the section 5 to explain the specification your files have to meet in order to have a desired output. If you respect the requirements of the section you are reading and the ones of the section 5, it is in the specification of `b2rust` that your components shall be translated into a compilable Rust program (eventually made up of several files).

4.1 `b2rust` calling options

Currently, `b2rust` only works on B "modules". If you want to use `b2rust`, you should be familiar with the B environment terminology. However, to put it in a nutshell, let's precise it is made up of an abstract machine (basically, a `.mch` file) and an implementation (basically, a `.imp` file). Yet, `b2rust` doesn't work on them directly; it uses BXML files. A BXML file is a more convenient shape of a B component which allows the developer not to have to parse a B component himself. It is a specific XML file which is automatically generated by Atelier B from a B component using the `bxml` tool. Then, depending on its calling parameters, `b2rust` shall generate a Rust program (figure 2) which can be written to `stdout` or to a file.

Also, know that the B method allows to work on several modules, as an implementation can *import* a module. However, as the importation diagram cannot contain any cycle (as long as your project is "verified"), there must be an "entry point", i.e. a module which recursively imports all the others. We refer it as the *entry module*.

So, `b2rust` must find the abstract machine and the implementation of the entry module and all recursively imported modules (the abstract machine is necessary to get more information on some clauses or expressions). Therefore, you need to provide the name of the B entry module and the directory we can find the files in. The syntax is the following:

```
b2rust module -I directory
   [-O output_directory]
```

- `module`: specify the B "entry module" name.
- `-I directory`: specify a directory where the BXML files associated to this module and whose name have to be `module.bxml` and `module_i.bxml` can be found; this directory should also include recursively imported modules, if any;
- `-O output_directory`: specify an output directory where `b2rust` shall

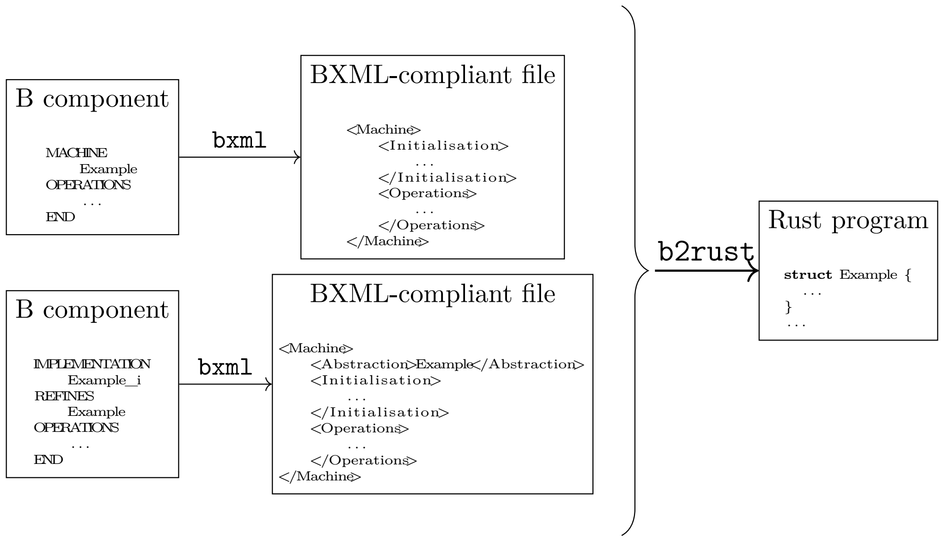

Figure 2: `bxml` generates a BXML file from a B component. `b2rust` generates a Rust code from two BXML files, which are searched in a given directory.

put the generated codes. Note that the translation of a module (`name.bxml`, `name_i.bxml`) will be a Rust code whose name will be `name.rs`. If no `output_directory` is provided, it will be set to `.`, the current directory.

Please note that the parameters can be given to `b2rust` in whatever order. However, no parameter shall be given twice! So:

```
b2rust module -I mydir -O here
```

does the same thing as:

```
b2rust -I mydir -O here module
```

, but:

```
b2rust -I mydir -O here module -O there
```

or:

```
b2rust module1 module2 -I mydir -O here
```

shall throw an error.

4.2 Errors

These two propositions are strictly equivalent:

1. The return code of the execution of `b2rust` is 0.

2. Rust code was generated by `b2rust`. The entry module code translation is compilable by `rustc` according the specification detailed in the section 4.3 and match the B specification of your components.

If an error occurs, a message will be printed on the `stderr` stream and, hopefully, will give you some precious information. However, it is not because a code was created that it is correct!

4.3 Usage of generated files

4.3.1 Context

The code `b2rust` generates (eventually made up of several files, if imported components) is a library code; it doesn't contain any `main` function. This allows the user to write his own `main`, which can be useful to specify the entry point of the program or to test the library on his own. A file containing only a `main` may then be compiled as long as the generated library path is specified.

So, `b2rust` doesn't compile the file itself and, if the code translation of the entry module is stored in a `a.rs` file, you can compile it with the following command:

```
$ rustc --crate-type=lib a.rs
```

This will generate a `liba.rlib` file which can be included in other projects. The `rustc` command calls the Rust compiler; it can probably be installed separately, or from `rustup`, the official Rust toolchain installer. Anyway, this subject is outside the scope of this document.

The user may then want to write his own `main` function (so, in Rust), and bind it to the compiled library. If his file is `main.rs`, then, the command

```
$ rustc --extern a=liba.rlib main.rs
```

will do the trick. Note, however, that a struct `struct` defined in the library needs to be referenced by the syntax `a::struct`.

If your project uses recursively imported modules, the latter commands are sufficient, as long as the other component code translations are in the same directory as `a.rs`.

4.3.2 Specification

All the following specifications must be respected. If they seem difficult to understand, and example is given in the section 4.3.3.

1. Every call to a library's function *must* follow the preconditions of the associated operation you may find in the abstract machine. No test is done automatically.

2. The only interface of the library you may use is the call to the functions. (Anyway, the Rust compiler should prevent any other access.)

3. A call to any struct instance's function needs to be preceded by a call to the `initialisation` procedure.

4. A B operation called f with the signature: $o_1, \cdots, o_n \leftarrow f(i_1, \cdots, i_m)$ is called with the syntax $f(i_1, \cdots, i_m, o_1, \cdots, o_n)$ (mind the order); however, input parameters need to be call by immutable reference to the associated Rust type and output parameters by mutable reference to the associated Rust type.

4.3.3 Recommandations

Before using an operation of the component, instanciate the component (which is translated by a `struct` in Rust):

```
let mut s: component::component = Default::default();
```

Note that:

1. The object has to be mutable (because the operation may modify the field of the struct);

2. `component` has to be replaced by the library's name (the name of the compiled crate); hint: if the library is `libfoo.rlib`, it should be `foo`;

3. The Rust struct (which matches a component) implements the `Default` trait, so, you may use the useful `default` function. As Rust refuses to call the `initialisation` if the object is not instaciated, it is even compulsory.

Then, before doing anything, you *must* run the initialisation:

```
s.initialisation();
```

Now, if we want to call an operation with this signature:

```
o1, o2, o3 <-- op(i1, i2, i3) = PRE
    o1 : rust_i32 &
    o2 : rust_i8 &
    o3 : rust_i16 &
    i1 : rust_bool &
```

```
    i2 : rust_i8 &
    i3 : rust_i16
THEN
```

The associated function `s.op` has six parameters matching the types specified in the specification; as the variable need to be instanciated first, your program might look like this:

```
let mut o1: i32 = 1234;
let mut o2: i8 = -67;
let mut o3: i16 = 0;
let i1: bool = false;
let i2: i8 = 12;
let i3: i16 = 0;
s.op(&i1, &i2, &i3, &mut o1, &mut o2, &mut o3);
```

As Rust surprisingly accepts references to litterals, the latter code could also be written:

```
let mut o1: i32 = 1234;
let mut o2: i8 = -67;
let mut o3: i16 = 0;
s.op(&false, &12, &0, &mut o1, &mut o2, &mut o3);
```

5 Application field of `b2rust`

Let us define $\mathsf{directory}_i, \mathsf{directory}_o, \mathsf{module}$ three characters strings given as parameter to `b2rust` (for example, with the call `b2rust -I` $\mathsf{directory}_i$ `-O` $\mathsf{directory}_o$ module). It is the goal of this section to explicit the requirements $\mathsf{directory}_i, \mathsf{directory}_o, \mathsf{module}$ need to meet in order to allow `b2rust` to generate a correct code, i.e. a Rust code usable by a final user without errors (but, concerning the Rust compilation, not necessarily without warnings) like specified section 4.3, and which matches the specification of the B programs.

The requirements are:

1. $\mathsf{directory}_i$ has to be a path to an existing directory in the filesystem and `b2rust` needs to have execution permissions on it (so that it can read its files);

2. The files $e_a \coloneqq \mathsf{directory}_{\mathtt{i}}/\mathsf{module}.\mathtt{bxml}$ and $e_i \coloneqq \mathsf{directory}_{\mathtt{i}}/\mathsf{module}\_\mathtt{i}.\mathtt{bxml}$ have to be existing files of the filesystem, and `b2rust` needs to have read access to e_a and e_i.

3. e_a and e_i must be BXML 1.0 compliant files.

4. e_a has to be an abstract machine.

5. e_i has to be the implementation of e_a.

6. Atelier B must be able to prove the e_a and e_i components.

7. e_i has to respect the "modules requirements" you can find section 5.1.

Before giving the next requirement, let us define $i_1, \cdots, i_n$ the characters strings which represent the machines imported by e_i. If there are none, $n = 0$. If e_i is associated to a B program which contains the following lines:

```
IMPORTS
    M1.module01,
    module03
    M2.module01,
    M3.module02
```

, then, $n = 4$ and $i_1 =$ `module01`, $i_2 =$ `module03`, $i_3 = i_1$ and $i_4 =$ `module02`.

8. For any $k \in \{1, \cdots, n\}$, the requirements 2 to 8 also apply if you replace e_a and e_i by $i_{k,a} \coloneqq \mathsf{directory_i}/i_k$`.bxml` and $i_{k,i} \coloneqq \mathsf{directory_i}/i_k$`__i.bxml`. (This also applies to the definition of sub-imports used in requirement 8, so, they apply to recursively imported modules.)

9. The Atelier B project which has (e_a, e_i) as entry module must be checked by the "Project checker".

For reasons which have not been yet fully understood, some nested expressions using maths operators and especially assignements are not well translated by `b2rust`. It might be because B authorizes assignements from 32-bits integer to 8-bits integer, for example, but Rust doesn't. Anyway — it should be corrected. Some tests fail and highlight this problem.

5.1 Modules requirements

Let us consider two standard BXML 1.0 files we call a' and i' which match the requirements 2 to 6 of the last section. They match B programs we call respectively a and i.

Let us define requirements a and i could respect:

allo i is the implementation of a;

$\mathcal{R}_1$ i doesn't `SEES`, `INCLUDE`, `EXTENDS`, `USES` any other components, **except** `b2rust_types`, a file you can find in the `files/` directory and which defines the types `b2rust` recognizes.

$\mathcal{R}_2$ i matches the grammar $\mathcal{G}_i$ whose definition follows:

$\mathcal{G}_i =$ `IMPLEMENTATION` Ident `_i`
`REFINES` Ident
`SEES` `b2rust_types`
ClauseImplantation$_{\mathtt{b2r}}$*
`END`

(The definition of ClauseImplantation$_{\mathtt{b2r}}$ is given later.)

$\mathcal{R}_3$ For each concrete constant we call c (c is a characters string) defined somewhere among the components, the `PROPERTIES` clause of i has to contain a Prédicat under the shape

c `:` RecognizedType

$\mathcal{R}_4$ For each concrete variable we call v (v is a characters string) defined somewhere among the components, the `INVARIANT` clause of i has to contain a Prédicat under the shape

v `:` RecognizedType

$\mathcal{R}_5$ For each usage of a LocalVariableInstruction, the introduced local variables have to be typed using a RecognizedType in the nested instructions using a Substitution_devient_elt_de$_{\mathtt{b2r}}$ instruction. More precisely, if a local variable instruction introduces n new variables whose names are $v_1, \cdots, v_n$, then:

(a) It needs to have at least n nested instructions.

(b) The first n nested instructions must follow the shape of SubstitutionDevientEltDe$_{\mathtt{b2r}}$, where the Ident is exactly the name of a declared local variable.

(c) Each declared local variable must be typed this way. As a consequence of $\mathcal{R}_5$b and $\mathcal{R}_6$, each declared local variable must be typed exactly once this way.

$\mathcal{R}_6$ Outside the circumstances defined under proposition $\mathcal{R}_5$, usage of the SubstitutionDevientEltDe$_{\mathtt{b2r}}$ is forbidden.

$\mathcal{R}_7$ For each operation of i, if we call n its number of input parameters and m its number of output parameters, then, if $n + m \geq 1$:

(a) The abstraction of the operation must have a precondition.

(b) This precondition must be under the shape $p_1 \boxed{\&} p_2 \boxed{\&} \cdots \boxed{\&} p_p$ (we might also have $p = 1$).

(c) For each input or output parameter named s[2], there must be exactly one predicate among $p_1, \cdots, p_p$ under the shape s $\boxed{:}$ RecognizedType. (So, do not type twice a parameter, even if these typings are not mutually exclusive. However, you may write $s\boxed{\texttt{:␣INT}}$, and, later on, $s\boxed{\texttt{:␣rust\_i32}}$, for example.)

The grammar $\mathcal{G}_i$ definition uses other grammars we give the definition below. They use explicit names (they correspond to known B clause names or known computer science entities), so we shall take the opportunity to explain precisely the subset of B0 `b2rust` translates.

To give the user clear information about the translation scope of `b2rust` from the point of view of B0, the B0 grammar recognition (given in the document *Manuel de référence du langage B*, but some terms are translated into English below) is specified on the left column in the following definitions. On the right, you can find the `b2rust` recognized grammar.

The grammar uses a syntax which heavily relies one regular expressions (*regexps*). In the following examples, e and f are regexps.

- $\boxed{\texttt{hello\textbackslash 1243\_?\{[}}$ matches exactly the character string `hello\1234_?{[`;
- A regular expression can be stored into a constant which is typeset with a sans serif font, in blue, and underlined, like this; if you read a numerized version of this document, it also creates a link to the definition (which is non valid in our example, but here is a valid version: ClauseInitialisationB0). A grey typeset (like this) just means it is a blind syntax, i.e. it is not defined in the document and it has no link;
- $e \cdot f$ matches exactly e followed by f (when possible, the dot is omitted);
- $e \mid f$ matches exactly e and f;
- $e+$ mateches exactly e or ee or eee, etc. In other terms, it matches the e regexp repeated anytime. Note that regexp can be aggregated using parenteses, like usual mathematical expressions;
- $e+^f$ matches e or efe or $efefe$, etc. In other terms, it matches the e regexp repeated anytime, but using f as separator;
- $e*$ is the same as $e+$, but it also matches nothing (i.e. the empty string characters). $e*^f$ is defined the same way;
- $e?$ matches e and nothing;

[2]Reminder: according to the B reference, the parameters name are the same between a machine's operation and its implementation.

- $[\boxed{\texttt{a}} - \boxed{\texttt{b}}]$ matches any character *between* $\boxed{\texttt{a}}$ and $\boxed{\texttt{b}}$, if this has any sense; it works only with digits (for example, $[\boxed{\texttt{2}} - \boxed{\texttt{9}}]$ matches $\boxed{\texttt{2}}$, $\boxed{\texttt{3}}$, etc., $\boxed{\texttt{9}}$) and letters ($[\boxed{\texttt{A}} - \boxed{\texttt{C}}]$ matches $\boxed{\texttt{A}}$, $\boxed{\texttt{B}}$, $\boxed{\texttt{C}}$, $[\boxed{\texttt{u}} - \boxed{\texttt{w}}]$ matches $\boxed{\texttt{u}}$, $\boxed{\texttt{v}}$, $\boxed{\texttt{w}}$).

5.2 Clauses

ClauseImplantation = ClauseSees |
ClauseImports |
ClausePromotes |
ClauseExtendsB0 |
ClauseSets |
ClauseConcreteConstants |
ClauseProperties |
ClauseValues |
ClauseConcreteVariables |
ClauseInvariant |
ClauseAssertions |
ClauseInitialisationB0 |
ClauseOperationsB0

ClauseImplantation$_{\texttt{b2r}}$ = ClauseSees |
ClauseImports$_{\texttt{b2r}}$ |
ClauseConcreteConstants |
ClauseProperties |
ClauseValues$_{\texttt{b2r}}$ |
ClauseConcreteVariables$_{\texttt{b2r}}$ |
ClauseInvariant |
ClauseAssertions |
ClauseInitialisationB0$_{\texttt{b2r}}$ |
ClauseOperationsB0$_{\texttt{b2r}}$

ClauseImports = $\boxed{\texttt{IMPORTS}}$ (IdentRen (InstanciationB0$+^{\boxed{,}}$)?) $+^{\boxed{,}}$

ClauseImports$_{\texttt{b2r}}$ = $\boxed{\texttt{IMPORTS}}$ (IdentRen) $+^{\boxed{,}}$

ClauseConcreteVariables = $\boxed{\texttt{CONCRETE\_VARIABLES}}$ IdentRen$*^{\boxed{,}}$

ClauseConcreteVariables$_{\texttt{b2r}}$ = $\boxed{\texttt{CONCRETE\_VARIABLES}}$ Ident$*^{\boxed{,}}$

ClauseValues = $\boxed{\texttt{VALUES}}$ Valuation$+^{\boxed{;}}$

ClauseValues$_{\texttt{b2r}}$ = $\boxed{\texttt{VALUES}}$ Instruction$*^{\boxed{;}}$

ClauseInitialisationB0 = $\boxed{\texttt{INITIALISATION}}$ Instruction$*^{\boxed{;}}$

ClauseInitialisationB0$_{\texttt{b2r}}$ = $\boxed{\texttt{INITIALISATION}}$ Instruction$_{\texttt{b2r}}*^{\boxed{;}}$

ClauseOperationsB0 = $\boxed{\texttt{OPERATIONS}}$ OpérationB0$*^{\boxed{;}}$

ClauseOperationsB0$_{\texttt{b2r}}$ = $\boxed{\texttt{OPERATIONS}}$ OpérationB0$_{\texttt{b2r}}*^{\boxed{;}}$

OpérationB0 = EntêteOpération $\boxed{\texttt{=}}$ Instruction$+^{\boxed{;}}$

OpérationB0$_{\texttt{b2r}}$ = EntêteOpération$_{\texttt{b2r}}$ $\boxed{\texttt{=}}$ Instruction$_{\texttt{b2r}}+^{\boxed{;}}$

EntêteOpération = (Ident$^{+\boxed{,}}$ $\boxed{\texttt{<--}}$)?IdentRen ($\boxed{\texttt{(}}$Ident $+^{\boxed{,}}$ $\boxed{\texttt{)}}$)?

EntêteOpération$_{\texttt{b2r}}$ = (Ident$^{+\boxed{,}}$ $\boxed{\texttt{<--}}$)?Ident ($\boxed{\texttt{(}}$Ident $+^{\boxed{,}}$ $\boxed{\texttt{)}}$)?

5.3 Instructions

Instruction = BlockInstruction |
LocalVariableInstruction |
IdentitySubstitution |
BecomesEqualInstruction |
OperationCallInstruction |
ConditionalInstruction |
CaseInstruction |
AssertionInstruction |
SequenceInstruction |
SubstitutionTantQue

$\text{Instruction}_{\text{b2r}}$ = $\text{LocalVariableInstruction}_{\text{b2r}}$ |
$\text{BecomesEqualInstruction}_{\text{b2r}}$ |
$\text{SubstitutionDevientEltDe}_{\text{b2r}}$ |
$\text{ConditionalInstruction}_{\text{b2r}}$ |
$\text{AssertionInstruction}_{\text{b2r}}$
$\text{SequenceInstruction}_{\text{b2r}}$ |
$\text{SubstitutionTantQue}_{\text{b2r}}$

LocalVariableInstruction = `VAR` $\text{Ident}^{+\,\texttt{,}}$ `IN` Instruction `END`

$\text{LocalVariableInstruction}_{\text{b2r}}$ = `VAR` $\text{Ident}^{+\,\texttt{,}}$ `IN` $\text{Instruction}_{\text{b2r}}$ `;` $\text{Instruction}_{\text{b2r}}$ `END`

It needs to have at least two nested instructions.

BecomesEqualInstruction
= IdentRen (`(` $\text{Terme}+^{\texttt{,}}$ `)`)? `:=` Terme |
IdentRen `:=` ExprTableau |
IdentRen (`'` Ident)+ `:=` Terme

$\text{BecomesEqualInstruction}_{\text{b2r}}$
= $\text{Ident}_{\text{b2r}}$ (`(` $\text{Terme}_{\text{b2r}}+^{\texttt{,}}$ `)`)? `:=` $\text{Terme}_{\text{b2r}}$ |
$\text{Ident}_{\text{b2r}}$ `:=` $\text{ExprTableau}_{\text{b2r}}$ |
$\text{Ident}_{\text{b2r}}$ `:=` $\text{Terme}_{\text{b2r}}$

$\text{SubstitutionDevientEltDe}_{\text{b2r}}$ = Ident `::` RecognizedType

This special substitution is technically not an instruction (i.e. it is not B0-compliant). However, it will be considered as an instruction in the following sections. For its usage, see the input requirement number $\mathcal{R}_5$.

$$\underline{\text{ConditionalInstruction}} = \boxed{\texttt{IF}}\underline{\text{Condition}}\ \boxed{\texttt{THEN}}\underline{\text{Instruction}}\ \left(\begin{array}{l}\boxed{\texttt{ELSIF}}\underline{\text{Condition}}\\ \boxed{\texttt{THEN}}\underline{\text{Instruction}}\end{array}\right)^{*}\ (\boxed{\texttt{ELSE}}\underline{\text{Instruction}})?\ \boxed{\texttt{END}}$$

$$\underline{\text{ConditionalInstruction}_{\texttt{b2r}}} = \boxed{\texttt{IF}}\underline{\text{Condition}_{\texttt{b2r}}}\ \boxed{\texttt{THEN}}\underline{\text{Instruction}_{\texttt{b2r}}}\ \left(\begin{array}{l}\boxed{\texttt{ELSIF}}\underline{\text{Condition}_{\texttt{b2r}}}\\ \boxed{\texttt{THEN}}\underline{\text{Instruction}_{\texttt{b2r}}}\end{array}\right)^{*}\ (\boxed{\texttt{ELSE}}\underline{\text{Instruction}_{\texttt{b2r}}})?\ \boxed{\texttt{END}}$$

$$\underline{\text{AssertionInstruction}} = \boxed{\texttt{ASSERT}}\ \underline{\text{Prédicat}}\ \boxed{\texttt{THEN}}\ \underline{\text{Instruction}}\ \boxed{\texttt{END}}$$

$$\underline{\text{AssertionInstruction}_{\texttt{b2r}}} = \boxed{\texttt{ASSERT}}\ \underline{\text{Prédicat}}\ \boxed{\texttt{THEN}}\ \underline{\text{Instruction}_{\texttt{b2r}}}\ \boxed{\texttt{END}}$$

The elements after the `INVARIANT` and `VARIANT` keyword concern the B proof and are just ignored when it comes to the translation. So, you can just put anything around them.

$$\underline{\text{SequenceInstruction}} = \underline{\text{Instruction}}\boxed{\texttt{;}}\ \underline{\text{Instruction}}$$

$$\underline{\text{SequenceInstruction}_{\texttt{b2r}}} = \underline{\text{Instruction}_{\texttt{b2r}}}\boxed{\texttt{;}}\ \underline{\text{Instruction}_{\texttt{b2r}}}$$

$$\underline{\text{SubstitutionTantQue}} = \boxed{\texttt{WHILE}}\underline{\text{Condition}}\boxed{\texttt{DO}}\ \underline{\text{Instruction}}\ \boxed{\texttt{INVARIANT}}\underline{\text{Prédicat}}\ \boxed{\texttt{VARIANT}}\underline{\text{Expression}}$$

$$\underline{\text{SubstitutionTantQue}_{\texttt{b2r}}} = \boxed{\texttt{WHILE}}\underline{\text{Condition}_{\texttt{b2r}}}\boxed{\texttt{DO}}\ \underline{\text{Instruction}_{\texttt{b2r}}}\ \boxed{\texttt{INVARIANT}}\underline{\text{Prédicat}}\ \boxed{\texttt{VARIANT}}\underline{\text{Expression}}$$

5.4 Condition

$$
\begin{aligned}
\text{Condition} = {} & \text{TermeSimple}\ \boxed{\texttt{=}}\ \text{TermeSimple} \mid \\
& \text{TermeSimple}\ \boxed{\texttt{/=}}\ \text{TermeSimple} \mid \\
& \text{TermeSimple}\ \boxed{\texttt{<}}\ \text{TermeSimple} \mid \\
& \text{TermeSimple}\ \boxed{\texttt{>}}\ \text{TermeSimple} \mid \\
& \text{TermeSimple}\ \boxed{\texttt{<=}}\ \text{TermeSimple} \mid \\
& \text{TermeSimple}\ \boxed{\texttt{>=}}\ \text{TermeSimple} \mid \\
& \text{Condition}\ \boxed{\texttt{\&}}\ \text{Condition} \mid \\
& \text{Condition}\ \boxed{\texttt{or}}\ \text{Condition} \mid \\
& \boxed{\texttt{not}}\boxed{\texttt{(}}\text{Condition}\boxed{\texttt{)}} \mid \\
& \boxed{\texttt{(}}\text{Condition}\boxed{\texttt{)}}
\end{aligned}
$$

$$
\begin{aligned}
\text{Condition}_{\texttt{b2r}} = {} & \text{TermeSimple}_{\texttt{b2r}}\ \boxed{\texttt{=}}\ \text{TermeSimple}_{\texttt{b2r}} \mid \\
& \text{TermeSimple}_{\texttt{b2r}}\ \boxed{\texttt{/=}}\ \text{TermeSimple}_{\texttt{b2r}} \mid \\
& \text{TermeSimple}_{\texttt{b2r}}\ \boxed{\texttt{<}}\ \text{TermeSimple}_{\texttt{b2r}} \mid \\
& \text{TermeSimple}_{\texttt{b2r}}\ \boxed{\texttt{>}}\ \text{TermeSimple}_{\texttt{b2r}} \mid \\
& \text{TermeSimple}_{\texttt{b2r}}\ \boxed{\texttt{<=}}\ \text{TermeSimple}_{\texttt{b2r}} \mid \\
& \text{TermeSimple}_{\texttt{b2r}}\ \boxed{\texttt{>=}}\ \text{TermeSimple}_{\texttt{b2r}} \\
& \text{Condition}_{\texttt{b2r}}\ \boxed{\texttt{\&}}\ \text{Condition}_{\texttt{b2r}} \mid \\
& \text{Condition}_{\texttt{b2r}}\ \boxed{\texttt{or}}\ \text{Condition}_{\texttt{b2r}} \mid \\
& \boxed{\texttt{not}}\boxed{\texttt{(}}\text{Condition}_{\texttt{b2r}}\boxed{\texttt{)}} \mid \\
& \boxed{\texttt{(}}\text{Condition}_{\texttt{b2r}}\boxed{\texttt{)}}
\end{aligned}
$$

5.5 Expressions

$$
\begin{aligned}
\text{ExprTableau} = {} & \text{Ident} \mid \\
& \boxed{\texttt{\{}}\left(\text{TermeSimple}^{+\boxed{\texttt{|->}}}\boxed{\texttt{|->}}\text{Terme}\right)^{+\boxed{\texttt{,}}}\boxed{\texttt{\}}} \mid \\
& \text{EnsembleSimple}^{+\boxed{\texttt{*}}}\boxed{\texttt{*\{}}\text{Terme}\boxed{\texttt{\}}}
\end{aligned}
$$

$$
\begin{aligned}
\text{ExprTableau}_{\texttt{b2r}} = {} & \text{Ident} \mid \\
& \boxed{\texttt{\{}}\left(\text{EntierLit}_{\texttt{b2r}}\boxed{\texttt{|->}}\left(\text{Terme} \mid \text{ExprTableau}_{\texttt{b2r}}\right)\right)^{+\boxed{\texttt{,}}}\boxed{\texttt{\}}}
\end{aligned}
$$

$$
\begin{aligned}
\text{TermeSimple} = {} & \text{IdentRen} \mid \\
& \text{EntierLit} \mid \\
& \text{BooléenLit} \mid \\
& \text{IdentRen}\ (\boxed{\texttt{'}}\ \text{Ident})+
\end{aligned}
$$

$$
\begin{aligned}
\text{TermeSimple}_{\texttt{b2r}} = {} & \text{EntierLit}_{\texttt{b2r}} \mid \\
& \text{BooléenLit} \mid \\
& \text{Ident}
\end{aligned}
$$

$$\text{IdentRen} = \text{Ident}^{+\boxed{\texttt{,}}}$$

$$\text{IdentRen}_{\texttt{b2r}} = \text{Ident}^{+\boxed{\texttt{,}}}$$

A renamed indentifier.

$$\text{Ident} = ([\boxed{\texttt{a}} - \boxed{\texttt{z}}] \mid [\boxed{\texttt{A}} - \boxed{\texttt{Z}}])\ ([\boxed{\texttt{a}} - \boxed{\texttt{z}}] \mid [\boxed{\texttt{A}} - \boxed{\texttt{Z}}] \mid [\boxed{\texttt{0}} - \boxed{\texttt{9}}] \mid \boxed{\texttt{\_}})*$$

$$\text{BooléenLit} = \boxed{\texttt{FALSE}} \mid \boxed{\texttt{TRUE}}$$

Terme = TermeSimple |
ExpressionArithmétique |
TermeRecord |
TermeRecord (`'` Ident) +

$\text{Terme}_{\texttt{b2r}}$ = $\text{TermeSimple}_{\texttt{b2r}}$ |
$\text{ExpressionArithmétique}_{\texttt{b2r}}$

ExpressionArithmétique = EntierLit |
IdentRen |
IdentRen `(` Terme +`,` `)` + |
IdentRen (`'` Ident) |
(ExpressionArithmétique [`+` `-` `*` `/` (`mod`) `**`] ExpressionArithmétique) |
`-` ExpressionArithmétique |
`succ(` ExpressionArithmétique `)` |
`pred(` ExpressionArithmétique `)` |
`(` ExpressionArithmétique `)`

$\text{ExpressionArithmétique}_{\texttt{b2r}}$ = $\text{EntierLit}_{\texttt{b2r}}$ |
Ident |
(ExpressionArithmétique [`+` `-` `*` `/` (`mod`) `**`] ExpressionArithmétique)

EntierLit = EntierLittéral |
`MAXINT` |
`MININT`

$\text{EntierLit}_{\texttt{b2r}}$ = $\text{EntierLittéral}_{\texttt{b2r}}$

EntierLittéral = `-`? $[0-9]+$

An EntierLittéral (a literal of integer) has to be $\geq$ `MININT` and $\leq$ `MAXINT`.

$\text{EntierLittéral}_{\texttt{b2r}}$ = `-`? $[0-9]+$

Digit = $[0-9]+$

5.6 Recognized types

RecognizedType = `rust_i32` | `rust_i16` | `rust_i8` | `rust_bool` |
`0..` Digit `-->␣(` RecognizedType `)`

Note that:

- The types you may find strange ($\boxed{\texttt{rust\_i32}}$, etc.) are Rust types defined in B; they are defined in a file your components have to `SEE`. More information on the Rust types can be found in the appendix.
- So, the only relations allowed in "BRust" are total function from an integer interval beginning with 0.
- If the right member is a simple recognized type, a version without the parentheses is also parsed.

6 Translation scheme

The role of this part is to specify the translation scheme of `b2rust`.

6.1 `b2rust` inner workings

Firstly, let us give some information. `b2rust` consists of a main source file, `b2rust.cpp`, which:

1. Creates an `Input` object from the command lines arguments. This is done because:
   (a) When a command line argument is needed, it shouldn't be the role of every function to parse the `argv` argument. Instead, it is better to just query an attribute of an "Input" object;
   (b) Not all functions should be able to access the command line arguments;
   (c) The despicable `const char*` arguments have to be converted into nice `string` elements (maybe see the development conventions section).

   If the `Input` succeeds, it gives the programmer a `Module` (an object which consist of two `XMLElement*` pointing to the BXML implementation and the BXML abstract file) for the entry module and a `map` of `Modules` for every imported module.
2. Translate the entry module into a `Parser` (an object which consist of a C++ representation of the machine and a C++ representation of the implementation) and the `map` of `Modules` into a `map` of `Parsers`. This is the *parsing*. It allows us to reduce the request to the XML library and not having to deal with the XML syntax which can be deconcerting. The parsing take care of taking all relevant information from the BXML

files and representing them into a C++ object. The translation must be straightforward and no structure optimization must be made (however, some tiny assumptions are made on the BXML input given they respect the B language). This allows us to write scalable parsing functions.

3. Calls the `Checker`. Its theoretical role is to prevent usage of `b2rust` for modules which do not follow the requirements of section 5.1, however, it does not completely works.

4. Translate the entry module parser into a `RustModule` (an object which represent a Rust module) and the `map` of `Parser`s into a `map` of `RustModule`s. This is the *conversion*. This is where the serious job is done; a `RustModule` needs to be architecturally extremely close to a Rust code.

5. Prints every `RustModule`. The structure of a `RustModule` should be defined so that its printing is extremely straightforward. It should only consist of going over all attributes and subobjects and using a print function on them. The only optimization allowed it that an attribute may be printed more than once, but even that should be avoided if this leads to a still reasonable structure. Every `RustModule` is printed into a different file whose name is the one of the module, plus "`.rs`"; so, **a B module is bijectively associated by exactly one produced code file**.

This document shan't specify precisely the translation scheme of all these steps; luckily (and purposely), the step 2 is straightforward enough we can omit its specification[3]. The step 3 does not infer with the translation (it just reads data and, maybe, stops the translation). The step 5 should be specified, however, it deals with "output files" and we need to think of a way to mathematically represent this.

So, to sum it up, we need to specify precisely the `RustModule` object and the steps done to convert a `Parser` to a `RustModule` and a `RustModule` to a file. We need to find a way to represent C++ objects and a file stream.

6.2 Mathematical definitions

To represent mathematically the translation operated by `b2rust`, we represent the BXML input files, the C++ converted object defined by the *converter* and the code output by mathematical entities we call *structures*. Let us give some definitions.

- $\mathbb{B} = \{\top, \bot\}$.

[3]Because the C++ representation of the BXML file is extremely straightforward, as explained above; however, you need to know some assumptions are made on the BXML input file structure, so, the C++ structure of a parsed object doesn't *fully* match the BXML specification. I think the BXML contains imprecisions and there's an appendix section dedicated to this.

- $\mathbb{S}$ is the set of all alphanumeric strings, plus the underscore and $\boxed{\hookleftarrow}$, the carriage return character which represents a new line. The elements of $\mathbb{S}$ are typeset using a box with a shadow or with a rounded box, without incidence of their definition ($\boxed{\texttt{\_grasshopper12}} = \boxed{\textsf{\_grasshopper12}} \in \mathbb{S}$, for example). The choice of the notation depends on the willing of the writer and on the practical object we represent by a mathematical string; if it is a computational string, we will likely use the box with a shadow; if it is a file or a C++ `enum`, we will likely use a rounded box. $\boxed{}$ (or $\boxed{}$) is an empty string.

- If $a_1 \in A_1, \cdots, a_n \in A_n$, $\{a_1, \cdots, a_n\}$ is just a mathematical set, so, it has no order ($\{1,2\} = \{2,1\}$) and the cardinal of a set is its number of different elements: $|\{1,2,1\}| = 2$. (The elements of a set can have different properties, for example, $\{3, \{\}\} \in \mathbb{P}(\mathbb{Z} \cup \mathbb{P}(\mathbb{Z}))$ is a valid set.)

  We say a set can be recursively defined, for example, if we define $S = \mathbb{P}(\mathbb{Z} \cup S)$, we have $\{1, 3, \{2, \{1,3\}\}\} \in S$.

- $(a_1, \cdots, a_n)$ is called a tuple. Contrarily to sets, a tuple is ordered ($(1,2) \neq (2,1)$). There's only one way to represent a tuple; its number of elements (which can be redundant) is always defined: $|(1,2,1)| = 3$.

  Note we also define the $\in$ and $\notin$ operators on a tuple, but their meaning is counter-intuitive; actually, if $a \in A$ and $b \in B$, we say that (a,b) is mathematically represented as $\{\{\{a\},1\},\{\{b\},2\}\}$. So, it is the reason why $a \in (a,b)$ is false, but $(\{\{a\},1\} \in (a,b)$ is true. This definition ensures that a tuple is ordered *and* can be defined from equals elements. We can however write $a \mathrel{\in\mkern-10mu\shortmid} (a,b)$: if t is a tuple and a and element,

  $$a \mathrel{\in\mkern-10mu\shortmid} t \iff \exists n \in \mathbb{N}, \{\{a\}, n\} \in t$$

  Note that if t and u are tuples, $t \cup u$ is not a tuple, but $a \mathrel{\in\mkern-10mu\shortmid} t \cup u$ makes sense like we wish it would.

  The same remark goes for the $\exists$ and $\forall$ quantifiers: so, $\forall k \in (1,2,1)$ iterates on three *different* elements, but $\forall k \mathrel{\in\mkern-10mu\shortmid} (1,2,1)$ only iterates on two.

  Finally, as a tuple is ordered, we can refer to its elements using an *index* (note that a tuple is ordered from 1). So, if $n \geq 25$ and $t = (a_7, a_9, a_1)$, $t_3 = a_1$ and $t_1 = a_7$.

- *Structures* are mathematical entities which can be used to represent documents, code, or C++ objects. $\mathcal{S}$ is the set of all structures. In a nutshell, a structure is a tuple indexed by strings (we say that the *keys* are strings), so, for example, $s = \{\{\{a_3\}, \boxed{\texttt{cat}}\}, \{\{a_9\}, \boxed{\texttt{dog}}\}\}$ is a structure. This allows us to write $s.\mathsf{cat} = a_3$ (note the key is typeset in sans serif and not anymore with a box). In a structure, all the keys are pairwise distinct. Like the tuples, we can write $a_9 \mathrel{\in\mkern-10mu\shortmid} s$ and we call the *data* of a structure s the set $\{d \mid d \mathrel{\in\mkern-10mu\shortmid} s\}$ (we can say that a_9 is a *data* of s, for instance).

Contrarily to tuples, **a structure is not ordered.** This is because we do not define an order in $\mathbb{S}$, so, we cannot know if $\boxed{\texttt{cat}} < \boxed{\texttt{dog}}$, for example.

A structure can be represented with brackets, with the syntax which follows. If $s_1, \cdots, s_n \in \mathbb{S}$, we can write

$$s = \begin{bmatrix} s_1 & : a_1, \\ \cdots, & \\ s_n & : a_n \end{bmatrix} = \{\{\{a_1\}, s_1\}, \cdots, \{\{a_n\}, s_n\}\}$$

We say this syntax uses *fields*: for example, $s_8 \ : a_8$ is a field. (There's a chariot return between each field in this example, but it is not mandatory.)

As a structure is not ordered, we could also write

$$s = \begin{bmatrix} s_n & : a_n, \\ \cdots, & \\ s_1 & : a_1 \end{bmatrix}$$

However, if a structure has only one field, the key can of course be implied:

$$s = \begin{bmatrix} \mathsf{animal} & : \boxed{\texttt{cat}} \end{bmatrix} = \begin{bmatrix} \boxed{\texttt{cat}} \end{bmatrix}$$

In this example, the data $\boxed{\texttt{cat}}$ can be represented by $s.\mathsf{animal}$, but let us admit we can also write $s\cdot$. This is because the name of the key is not relevant when we deal with struct with an unique field.

- Now, it would be fine if we could define *sets* of structures. A set of structures is often typeset under the shape $\mathcal{S}.$ or under the shape Type (which defines a link in the numeric version of this document) and its definition uses doubled brackets and *fields definition* which use the $:\in$ operator:

$$\mathcal{S}_s = \left[\!\!\left[\begin{matrix} s_1 & :\in A_1, \\ \cdots, & \\ s_n & :\in A_n \end{matrix} \right]\!\!\right] = \{\{\{\{\alpha_1\}, s_1\}, \cdots, \{\{\alpha_n\}, s_n\}\} | \alpha_1 \in A_1, \cdots, \alpha_n \in A_n\}$$

Remark: the sets used in definition of sets of structures can be everything (including other sets of structures). So, if we use Cartesian products, we can refer to elements easily: for example, if $k, k' \in \mathbb{S}$,

$$\mathcal{S}_s = \left[\!\!\left[\begin{matrix} k & :\in A_1^n, \\ k' & :\in A_2 \end{matrix} \right]\!\!\right]$$

and $s \in \mathcal{S}_s$, we could write $s.k'$ as well as $s.k_8$ (or even $s.k$). (As $s.k \in A_1^n$, it is a tuple, so, its elements are ordered.)

During a set of structure definition, if a key is not specified, consider the key is anything different that the ones already in use (which is possible as a structure has always a countable number of fields).

$$\begin{bmatrix} \mathsf{cat} & : 13, \\ \mathsf{dog} & : \boxed{\texttt{parrot}}, \\ \boxed{\texttt{grasshopper}} & \end{bmatrix}$$

- If A is a set, $A^\infty = \bigcup_{k \in \mathbb{N}} A^k$. Note that if $a \in A^\infty$ and $|a| = 9$, for instance, $a \in A^9$ but $a \notin A^8$ and $a \notin A^{10}$.
- A structure can be quite useful to represent mathematically some concrete data (as C++ objects which have most of the time attributes which can be referred thanks to structures), but its definition can be quite troublesome when we want to represent documents or character strings which match a specific grammar. This is because the data of a structure are not *ordered*, so, there is not an unique way to represent it, and there's not an unique way to print a structure, for instance. This is the reason why we define now *ordered structures*. Mathematically, an order structure is an element of $\mathcal{S} \times \mathbb{P}\left(\mathbb{S}^2\right)$. An element of $\mathbb{S}^2$ defines an order between two keys. So, an element of $\mathbb{P}\left(\mathbb{S}^2\right)$ is just a set of key orders. An unordered structure like the ones we used previously can be seen as an ordered structure which has no key order defined, i.e. an element of $\mathcal{S} \times \{\} \subset \mathcal{S} \times \mathbb{P}\left(\mathbb{S}^2\right)$. Here is an example of an ordered structure which is not an unordered structure:

$$\left(\begin{array}{c} \left\{\{\{12\}, \boxed{\texttt{cat}}\}, \left\{\{24\}, \boxed{\texttt{dog}}\right\}, \left\{\{23\}, \boxed{\texttt{parrot}}\right\}, \left\{\{127\}, \boxed{\texttt{grasshopper}}\right\}\right\}, \\ \left\{\left(\boxed{\texttt{grasshopper}}, \boxed{\texttt{cat}}\right), \left(\boxed{\texttt{grasshopper}}, \boxed{\texttt{dog}}\right)\right\} \end{array} \right)$$

  In this example, the data 127 (associated to the key $\boxed{\texttt{grasshopper}}$) is defined as being *before* the data 12 or 24.

  The syntax used later is also used for ordered structures, but it can make use another operator: $\rightarrow$, which can replace the coma. It means the ordered structure specifies an order between the two keys. If we take the last example and replace $\left(\boxed{\texttt{grasshopper}}, \boxed{\texttt{dog}}\right)$ with $\left(\boxed{\texttt{cat}}, \boxed{\texttt{dog}}\right)$, the ordered structure can now be represented like this:

$$\begin{bmatrix} \mathsf{grasshopper} & : 127 \rightarrow \\ \mathsf{cat} & : 12 \rightarrow \\ \mathsf{dog} & : 24, \\ \mathsf{parrot} & : 23 \end{bmatrix}$$

  The syntax used for structures set is also expanded the same way to define ordered structures sets:

$$\left[\!\!\left[\begin{array}{ll} s_2 & :\in A_2, \\ s_1 & :\in A_1 \rightarrow \\ s_3 & :\in A_3, \end{array}\right]\!\!\right] = \left\{ \left(\begin{array}{c} \{\{\{\alpha_2\}, s_2\}, \{\{\alpha_1\}, s_1\}, \{\{\alpha_3\}, s_3\}\}, \\ \{(\alpha_1, \alpha_3)\} \end{array} \right) \middle| \alpha_1 \in A_1, \alpha_2 \in A_2, \alpha_3 \in A_3 \right\}$$

You might have noticed that the definition of an ordered structure "breaks" the $|\cdot|$ operator. So, if s is an ordered structure, we define $\|s\| = |s_1|$ which matches with what we want.

Also, if s is an ordered structure, its first element, s_1 is an unordered structure which has exactly the same fields. The notation $s\rangle$, more explicit than s_1, can be used.

- An ordered structure is told *totally ordered* if the orders definition "cover each key and forms a chain", or if it can be represented with the bracket syntax without commas but arrows. (This specific structure can be used to represent a document or a character string, for instance.) We define a specific syntax to allow the user not to use arrows everywhere:

$$\left[\!\!\left[\begin{array}{ll} s_1 & :\in A_1 \rightarrow \\ s_2 & :\in A_2 \rightarrow \\ s_3 & :\in A_3 \end{array}\right]\!\!\right] = \Downarrow \begin{array}{ll} s_1 & :\in A_1, \\ s_2 & :\in A_2, \\ s_3 & :\in A_3 \end{array} \Downarrow$$

$$\left[\begin{array}{ll} s_1 & : a_1 \rightarrow \\ s_2 & : a_2 \rightarrow \\ s_3 & : a_3 \end{array}\right] = \downarrow \begin{array}{ll} s_1 & : a_1, \\ s_2 & : a_2, \\ s_3 & : a_3 \end{array} \downarrow = \downarrow a_1, a_2, a_3 \downarrow$$

We define by $\mathcal{S}_\rightarrow$ the set of totally ordered structures.

Now that everything has been defined, let us define the translation scheme. `b2rust` takes as input BXML files we represent as ordered structures of an ordered structure set, $\mathcal{S}_{\text{BXML}}$. It translates them into C++ objects of type `RustModule` we represent as elements of $\mathcal{S}_{\text{converted}}$. For each `RustModule`, it then prints a code which is represented as an element of $\mathcal{S}_{\text{Rust}}$.

6.3 $\mathcal{S}_{\text{BXML}}$

$$\mathcal{S}_{\text{BXML}} = \Downarrow \begin{array}{ll} \textsf{Abstraction} & :\in \mathbb{S}, \\ \textsf{Imports} & :\in \underline{\textsf{instance\_list\_type}}, \\ \textsf{Values} & :\in \underline{\textsf{Valuation}}^\infty, \\ \textsf{Concrete\_Constants} & :\in \underline{\textsf{Id}}^\infty, \\ \textsf{Concrete\_Variables} & :\in \underline{\textsf{Id}}^\infty, \\ \textsf{Properties} & :\in \underline{\textsf{pred\_group}}, \\ \textsf{Invariant} & :\in \underline{\textsf{pred\_group}}, \\ \textsf{Initialisation} & :\in \underline{\textsf{Sub}}, \\ \textsf{Operations} & :\in \underline{\textsf{Operation}}^\infty \end{array} \Downarrow$$

$$\underline{\text{instance\_list\_type}} = \left\llbracket \text{Referenced\_Machine} \;:\in \underline{\text{Referenced\_Machine}}^{\infty} \right\rrbracket$$

$$\underline{\text{Referenced\_Machine}} = \left\Downarrow \begin{array}{ll} \text{Name} & :\in \mathbb{S}, \\ \text{Instance} & :\in \mathbb{S} \end{array} \right\Downarrow$$

$$\underline{\text{Valuation}} = \left\llbracket \begin{array}{ll} \text{ident} & :\in \mathbb{S}, \\ \text{Exp} & :\in \underline{\text{Exp}}^{\infty} \end{array} \right\rrbracket$$

$$\underline{\text{Exp}} = \begin{array}{lllll} \underline{\text{Unary\_Exp}} & \cup\underline{\text{Binary\_Exp}} & \cup\underline{\text{Ternary\_Exp}} & \cup\underline{\text{Nary\_Exp}} & \cup \\ \underline{\text{Boolean\_Literal}} & \cup\underline{\text{Boolean\_Exp}} & \cup\underline{\text{EmptySet}} & \cup\underline{\text{EmptySeq}} & \cup \\ \underline{\text{Id}} & \cup\underline{\text{Integer\_Literal}} & \cup\underline{\text{Quantified\_Exp}} & \cup\underline{\text{Quantified\_Set}} & \cup \\ \underline{\text{STRING\_Literal}} & \cup\underline{\text{Struct}} & \cup\underline{\text{Record}} & \cup\underline{\text{Real\_Literal}} & \cup \\ \underline{\text{Record\_Field\_Access}} & & & & \end{array}$$

$$\underline{\text{Binary\_Exp}} = \left\Downarrow \begin{array}{ll} \text{Exp} & :\in \underline{\text{Exp}}^{2}, \\ \text{op} & :\in \underline{\text{binary\_exp\_op}} \end{array} \right\Downarrow$$

$$\underline{\text{binary\_exp\_op}} = \left\{ \begin{array}{llllllllllll} \boxed{\texttt{,}}, & \boxed{\texttt{*}}, & \boxed{\texttt{*i}}, & \boxed{\texttt{*r}}, & \boxed{\texttt{*f}}, & \boxed{\texttt{*s}}, & \boxed{\texttt{**}}, & \boxed{\texttt{**i}}, & \boxed{\texttt{**r}}, & \boxed{\texttt{+}}, & \boxed{\texttt{+i}}, & \boxed{\texttt{+r}}, \\ \boxed{\texttt{+f}}, & \boxed{\texttt{+->}}, & \boxed{\texttt{+->>}}, & \boxed{\texttt{-}}, & \boxed{\texttt{-i}}, & \boxed{\texttt{-r}}, & \boxed{\texttt{-f}}, & \boxed{\texttt{-s}}, & \boxed{\texttt{-->}}, & \boxed{\texttt{-->>}}, & \boxed{\texttt{->}}, & \boxed{\texttt{..}}, \\ \boxed{\texttt{/}}, & \boxed{\texttt{/i}}, & \boxed{\texttt{/r}}, & \boxed{\texttt{/f}}, & \boxed{\texttt{/}\backslash}, & \boxed{\texttt{/|}\backslash}, & \boxed{\texttt{;}}, & \boxed{\texttt{<+}}, & \boxed{\texttt{<->}}, & \boxed{\texttt{<-}}, & \boxed{\texttt{<<|}}, & \boxed{\texttt{<|}}, \\ \boxed{\texttt{>+>}}, & \boxed{\texttt{>->}}, & \boxed{\texttt{>+>>}}, & \boxed{\texttt{>->>}}, & \boxed{\texttt{><}}, & \boxed{\texttt{||}}, & \boxed{\backslash\texttt{/}}, & \boxed{\backslash\texttt{|/}}, & \boxed{\texttt{\^{}}}, & \boxed{\texttt{mod}}, & \boxed{\texttt{|->}}, & \boxed{\texttt{|>}}, \\ \boxed{\texttt{|>>}}, & \boxed{\texttt{[}}, & \boxed{\texttt{(}}, & \boxed{\texttt{<'}}, & \boxed{\texttt{prj1}}, & \boxed{\texttt{prj2}}, & \boxed{\texttt{iterate}}, & \boxed{\texttt{const}}, & \boxed{\texttt{rank}}, & \boxed{\texttt{father}}, & \boxed{\texttt{subtree}}, & \boxed{\texttt{arity}} \end{array} \right\}$$

$$\underline{\text{Boolean\_Literal}} = \left\llbracket \text{value} \;:\in \mathbb{B} \right\rrbracket$$

$$\underline{\text{Id}} = \left\llbracket \text{value} \;:\in \mathbb{S} \right\rrbracket$$

$$\underline{\text{Integer\_Literal}} = \left\llbracket \text{value} \;:\in \mathbb{Z} \right\rrbracket$$

$$\underline{\text{Nary\_Exp}} = \left\Downarrow \begin{array}{ll} \text{exp} & :\in \underline{\text{Exp}}^{\infty}, \\ \text{op} & :\in \underline{\text{nary\_exp\_op}} \end{array} \right\Downarrow$$

$$\underline{\text{nary\_exp\_op}} = \boxed{\texttt{[}} \cup \boxed{\texttt{\{}}$$

$$\underline{\text{pred\_group}} = \underline{\text{Binary\_Pred}} \cup \underline{\text{Exp\_Comparison}} \cup \underline{\text{Quantified\_Pred}} \cup \underline{\text{Unary\_Pred}} \cup \underline{\text{Nary\_Pred}}$$

$$\underline{\text{Unary\_Pred}} = \left\llbracket \begin{array}{ll} \text{pred\_group} & :\in \underline{\text{pred\_group}}, \\ \text{op} & :\in \underline{\text{unary\_pred\_op}} \end{array} \right\rrbracket$$

$$\underline{\text{unary\_pred\_op}} = \{\boxed{\texttt{not}}\}$$

$$\underline{\text{Nary\_Pred}} = \left\llbracket \begin{array}{ll} \text{pred\_group} & :\in \underline{\text{pred\_group}}^{\infty}, \\ \text{op} & :\in \underline{\text{nary\_pred\_op}} \end{array} \right\rrbracket$$

$$\mathsf{nary\_pred\_op} = \{\boxed{\texttt{\&}}, \boxed{\texttt{or}}\}$$

$$\mathsf{Exp\_Comparison} = \Downarrow \begin{array}{lll} \mathsf{Exp} & :\in & \mathsf{Exp}^2, \\ \mathsf{op} & :\in & \mathsf{comparison\_op} \end{array} \Downarrow$$

$$\mathsf{comparison\_op} = \left\{ \begin{array}{l} \boxed{\texttt{:}}, \boxed{\texttt{/:}}, \boxed{\texttt{<:}}, \boxed{\texttt{/<:}}, \boxed{\texttt{<<:}}, \boxed{\texttt{/<<:}}, \boxed{\texttt{=}}, \boxed{\texttt{/=}}, \boxed{\texttt{>=}}, \boxed{\texttt{<}}, \boxed{\texttt{<=}}, \boxed{\texttt{>=i}}, \\ \boxed{\texttt{>i}}, \boxed{\texttt{<i}}, \boxed{\texttt{<=i}} \end{array} \right\}$$

$$\mathsf{Sub} = \begin{array}{llll} \mathsf{Bloc\_Sub} & \cup \boxed{\mathsf{Skip}} & \cup \mathsf{Assert\_Sub} & \cup \\ \mathsf{If\_Sub} & \cup \mathsf{Becomes\_Such\_That} & \cup \mathsf{Assignement\_Sub} & \cup \\ \mathsf{Select} & \cup \mathsf{Case\_Sub} & \cup \mathsf{ANY\_Sub} & \cup \\ \mathsf{LET\_Sub} & \cup \mathsf{VAR\_IN} & \cup \mathsf{Nary\_Sub} & \cup \\ \mathsf{Operation\_Call} & \cup \mathsf{Becomes\_In} & \cup \mathsf{While} & \cup \\ \mathsf{Witness} & & & \end{array}$$

$$\mathsf{Assert\_Sub} = \Downarrow \begin{array}{lll} \mathsf{Guard} & :\in & \mathsf{predicate\_type}, \\ \mathsf{Body} & :\in & \mathsf{substitution\_type} \end{array} \Downarrow$$

$$\mathsf{Nary\_Sub} = \left[\!\!\left[\begin{array}{lll} \mathsf{Sub} & :\in & \mathsf{Sub}^\infty, \\ \mathsf{op} & :\in & \mathsf{nary\_sub\_op} \end{array} \right]\!\!\right]$$

For this one, `outputParameters` are supposed to be Ids and not Exps (see section 9.3).

$$\mathsf{Operation\_Call} = \left[\!\!\left[\begin{array}{lll} \mathsf{name} & :\in & \left[\!\!\left[\mathsf{id} :\in \left[\!\!\left[\begin{array}{lll} \mathsf{value} & :\in & \mathbb{S}, \\ \mathsf{instance} & :\in & \mathbb{S}, \\ \mathsf{component} & :\in & \mathbb{S} \end{array} \right]\!\!\right] \right]\!\!\right], \\ \mathsf{inputParameters} & :\in & \left[\!\!\left[\mathsf{exp} :\in \mathsf{Exp}^\infty \right]\!\!\right], \\ \mathsf{outputParameters} & :\in & \left[\!\!\left[\mathsf{id} :\in \mathsf{Id}^\infty \right]\!\!\right] \end{array} \right]\!\!\right]$$

$$\mathsf{nary\_sub\_op} = \left\{\boxed{\texttt{||}}, \boxed{\texttt{;}}, \boxed{\texttt{CHOICE}}\right\}$$

$$\mathsf{If\_Sub} = \left[\!\!\left[\begin{array}{lll} \mathsf{Condition} & :\in & \mathsf{predicate\_type} \rightarrow \\ \mathsf{Then} & :\in & \mathsf{substitution\_type} \rightarrow \\ \mathsf{Else} & :\in & \mathsf{substitution\_type}, \\ \mathsf{elseif} & :\in & \mathbb{S} \end{array} \right]\!\!\right]$$

$$\mathsf{predicate\_type} = \mathsf{pred\_group}$$

$$\mathsf{substitution\_type} = \mathsf{Sub}$$

$$\mathsf{Assignement\_Sub} = \Downarrow \begin{array}{lll} \mathsf{Variables} & :\in & \mathsf{Exp}^\infty, \\ \mathsf{Values} & :\in & \mathsf{Exp}^\infty \end{array} \Downarrow$$

$$\mathsf{Operation} = \left\Downarrow \begin{array}{ll} \mathsf{OutputParameters} & :\in \mathsf{Id}^{\infty}, \\ \mathsf{InputParameters} & :\in \mathsf{Id}^{\infty}, \\ \mathsf{Precondition} & :\in \mathsf{pred\_group}, \\ \mathsf{Body} & :\in \mathsf{Sub}, \\ \mathsf{name} & :\in \mathbb{S} \end{array} \right\Downarrow$$

$$\mathsf{VAR\_IN} = \left\Downarrow \begin{array}{ll} \mathsf{Variables} & :\in \mathsf{variables\_type}, \\ \mathsf{Body} & :\in \mathsf{substitution\_type} \end{array} \right\Downarrow$$

$$\mathsf{While} = \left\Downarrow \begin{array}{ll} \mathsf{condition} & :\in \mathsf{predicate\_type}, \\ \mathsf{body} & :\in \mathsf{substitution\_type}, \\ \mathsf{invariant} & :\in \mathsf{predicate\_type}, \\ \mathsf{variant} & :\in \mathsf{expression\_type}, \end{array} \right\Downarrow$$

$$\mathsf{expression\_type} = \left[\, \mathsf{exp} \; :\in \mathsf{Exp} \, \right]$$

$$\mathsf{variables\_type} = \left[\!\left[\, \mathsf{\_Id} \; :\in \mathsf{Id}^{\infty} \, \right]\!\right]$$

$$\mathsf{Becomes\_In} = \left\Downarrow \begin{array}{ll} \mathsf{Variables} & :\in \mathsf{variables\_type}, \\ \mathsf{Value} & :\in \mathsf{expression\_type} \end{array} \right\Downarrow$$

$$\mathsf{expression\_type} = \left[\!\left[\, \mathsf{\_exp} \; :\in \mathsf{Exp} \, \right]\!\right]$$

6.4 $\mathcal{S}_{\text{converted}}$

$$\mathcal{S}_{\text{converted}} = \left[\!\left[\begin{array}{ll} \mathsf{mods} & :\in \mathbb{S}^{\infty}, \\ \mathsf{uses} & :\in \mathbb{S}^{\infty}, \\ \mathsf{name} & :\in \mathbb{S}, \\ \mathsf{instances} & :\in (\mathbb{S} \times \mathbb{S})^{\infty}, \\ \mathsf{variables} & :\in \mathbb{P}(\mathbb{S} \times \mathsf{Type}), \\ \mathsf{instances\_init} & :\in \mathbb{S}^{\infty}, \\ \mathsf{values} & :\in \mathsf{RustInstruction}^{\infty}, \\ \mathsf{initialisations} & :\in \mathsf{RustInstruction}^{\infty}, \\ \mathsf{functions} & :\in \mathsf{Function}^{\infty} \end{array} \right]\!\right]$$

$$\mathsf{Type} = \{\boxed{\mathsf{i32\_t}}, \boxed{\mathsf{i16\_t}}, \boxed{\mathsf{i8\_t}}, \boxed{\mathsf{bool\_t}}\} \cup \mathsf{tabular}$$

$$\mathsf{tabular} = \left[\!\left[\begin{array}{ll} \mathsf{size} & :\in \mathbb{Z}^{*}, \\ \mathsf{elementsType} & :\in \mathsf{Type}, \end{array} \right]\!\right]$$

Represents a tabular type.

$$\mathsf{RustInstruction} = \mathsf{RustAssignement} \cup \mathsf{RustIf} \cup \mathsf{Block} \cup \mathsf{Declaration} \cup \mathsf{FunctionCall} \cup \mathsf{RustWhile}$$

$$\underline{\mathsf{RustWhile}} = \left[\!\left[\begin{array}{ll} \mathsf{condition} & :\in \underline{\mathsf{RustPredicate}}, \\ \mathsf{instructions} & :\in \underline{\mathsf{RustInstruction}}^{\infty} \end{array} \right]\!\right]$$

$$\underline{\mathsf{FunctionCall}} = \left[\!\left[\begin{array}{ll} \mathsf{moduleName} & :\in \mathbb{S}, \\ \mathsf{functionName} & :\in \mathbb{S}, \\ \mathsf{inputParameters} & :\in \underline{\mathsf{RustExpression}}^{\infty}, \\ \mathsf{outputParameters} & :\in \underline{\mathsf{RustExpression}}^{\infty}, \end{array} \right]\!\right]$$

$$\underline{\mathsf{Declaration}} = \left[\!\left[\begin{array}{ll} \mathsf{name} & :\in \mathbb{S}, \\ \mathsf{type} & :\in \underline{\mathsf{Type}} \end{array} \right]\!\right]$$

Declaration of a variable. Useful, for example, to translate local variables.

$$\underline{\mathsf{Block}} = \left[\!\left[\; \mathsf{instructions} \; :\in \underline{\mathsf{RustInstruction}}^{\infty} \; \right]\!\right]$$

It just consist of nested instructions. Useful, for example, to translate local variables definition.

$$\underline{\mathsf{RustAssignement}} = \left[\!\left[\begin{array}{ll} \mathsf{variable} & :\in \underline{\mathsf{Variable}}, \\ \mathsf{expression} & :\in \underline{\mathsf{RustExpression}} \end{array} \right]\!\right]$$

$$\underline{\mathsf{RustExpression}} = \underline{\mathsf{Int}} \cup \underline{\mathsf{Bool}} \cup \underline{\mathsf{Variable}} \cup \underline{\mathsf{RustBinaryExpression}} \cup \underline{\mathsf{RustArray}}$$

$$\underline{\mathsf{Int}} = \left[\!\left[\; \mathsf{value} \; :\in \mathbb{Z} \; \right]\!\right] = \underline{\mathsf{Integer\_Literal}}$$

$$\underline{\mathsf{Bool}} = \left[\!\left[\; \mathsf{value} \; :\in \mathbb{B} \; \right]\!\right]$$

$$\underline{\mathsf{Variable}} = \underline{\mathsf{LocalVariable}} \cup \underline{\mathsf{GlobalVariable}} \cup \underline{\mathsf{ParameterVariable}}$$

$$\underline{\mathsf{LocalVariable}} = \left[\!\left[\; \mathsf{name} \; :\in \mathbb{S} \; \right]\!\right]$$

$$\underline{\mathsf{GlobalVariable}} = \left[\!\left[\; \mathsf{name} \; :\in \mathbb{S} \; \right]\!\right]$$

$$\underline{\mathsf{ParameterVariable}} = \left[\!\left[\; \mathsf{name} \; :\in \mathbb{S} \; \right]\!\right]$$

A variable which is a parameter of the function it is defined in. If they are translated by references, a variable usage should be dereferenced.

$$\underline{\mathsf{RustArray}} = \left[\!\left[\; \mathsf{values} \; :\in \underline{\mathsf{RustExpression}}^{\infty} \; \right]\!\right]$$

$$\underline{\mathsf{RustBinaryExpression}} = \left[\!\left[\begin{array}{ll} \mathsf{left\_expr} & :\in \underline{\mathsf{RustExpression}}, \\ \mathsf{type} & :\in \underline{\mathsf{binaryExpression}}, \\ \mathsf{right\_expr} & :\in \underline{\mathsf{RustExpression}} \end{array} \right]\!\right]$$

$$\underline{\text{binaryExpression}} = \{\boxed{\text{addition}}, \boxed{\text{substraction}}, \boxed{\text{division}}, \boxed{\text{multiplication}}, \boxed{\text{exponentiation}}, \boxed{\text{modulo}}, \boxed{\text{tabularPosition}}\}$$

$$\underline{\text{RustIf}} = \left[\!\left[\begin{array}{ll} \text{predicate} & :\in \underline{\text{RustPredicate}}, \\ \text{then\_instr} & :\in \underline{\text{RustInstruction}}^{\infty}, \\ \text{else\_instr} & :\in \underline{\text{RustInstruction}}^{\infty} \end{array} \right]\!\right]$$

$$\underline{\text{RustPredicate}} = \underline{\text{RustBinaryPredicate}} \cup \underline{\text{RustPredicateAggregate}} \cup \underline{\text{RustUnaryPredicate}}$$

$$\underline{\text{RustBinaryPredicate}} = \left[\!\left[\begin{array}{ll} \text{left\_expr} & :\in \underline{\text{RustExpression}}, \\ \text{symbol} & :\in \underline{\text{comparisonSymbol}}, \\ \text{right\_expr} & :\in \underline{\text{RustExpression}} \end{array} \right]\!\right]$$

$$\underline{\text{comparisonSymbol}} = \{\boxed{\text{LessThan}}, \boxed{\text{LessThanOrEqualTo}}, \boxed{\text{EqualTo}}, \boxed{\text{NotEqualTo}}, \boxed{\text{MoreThanOrEqualTo}}, \boxed{\text{MoreThan}}, \boxed{\text{And}}, \boxed{\text{Or}}\}$$

$$\underline{\text{RustPredicateAggregate}} = \left[\!\left[\begin{array}{ll} \text{predicates} & :\in \underline{\text{RustPredicate}}^{\infty}, \\ \text{symbol} & :\in \underline{\text{comparisonSymbol}} \end{array} \right]\!\right]$$

$$\underline{\text{RustUnaryPredicate}} = \left[\!\left[\text{predicate} \ :\in \underline{\text{RustPredicate}} \right]\!\right]$$

$$\underline{\text{Function}} = \left[\!\left[\begin{array}{ll} \text{name} & :\in \mathbb{S}, \\ \text{inputParameters} & :\in \mathbb{P}(\mathbb{S} \times \underline{\text{Types}}), \\ \text{outputParameters} & :\in \mathbb{P}(\mathbb{S} \times \underline{\text{Types}}), \\ \text{instructions} & :\in \underline{\text{RustInstruction}}^{\infty} \end{array} \right]\!\right]$$

$$\underline{\text{Context}} = \left[\!\left[\begin{array}{ll} \text{parameters} & :\in \mathbb{S}^{\infty}, \\ \text{global} & :\in \mathbb{S}^{\infty}, \\ \text{instancesNameAssoc} & :\in (\mathbb{S} \times \mathbb{S})^{\infty}, \\ \text{operationNameAssoc} & :\in (\mathbb{S} \times \mathbb{S})^{\infty} \end{array} \right]\!\right]$$

The context helps during the translation to keep an eye on several relevant information translated before the current translation unit (for example, list of local defined variables, list of instance names in `b2rust`...)

6.5 $\mathcal{S}_{\text{Rust}}$

$\mathcal{S}_{\text{Rust}} =$

mods :∈ _mod$^\infty$,
uses :∈ _use$^\infty$,
{ `#[derive(Default)]⤸`
`pub␣struct␣` },
name$_1$:∈ $\mathbb{S}$,
{ `␣{⤸` ,
`//␣Instances␣of␣imported␣modules.⤸` },
instances :∈ _instance$^\infty$,
{ `⤸//␣Concrete␣variables␣&␣constants.⤸` },
variables :∈ _variableDeclaration$^\infty$,
{ `}⤸⤸impl␣` },
name$_2$:∈ $\mathbb{S}$,
{ `␣{⤸`
`pub␣fn␣initialisation(&mut␣self)␣{⤸` ,
`//␣Instances␣of␣imported␣modules␣initialization.⤸` },
instances_init :∈ _instance_init$^\infty$,
{ `⤸//␣Constant's␣`VALUES`.⤸` },
values :∈ _instruction$^\infty$,
{ `⤸//␣`INITIALISATION`␣clause.⤸` },
initialisations :∈ _instruction$^\infty$,
{ `}⤸` },
functions :∈ _function$^\infty$,
{ `}⤸` }

_mod =

{ `mod␣` },
name :∈ $\mathbb{S}$,
{ `;⤸` }

_use =

{ `use␣` },
name :∈ $\mathbb{S}$,
{ `;⤸` }

_instance =

instanceName :∈ $\mathbb{S}$,
{ `:␣` },
name$_1$:∈ $\mathbb{S}$,
{ `::` },
name$_2$:∈ $\mathbb{S}$,
{ `,⤸` }

The prefix $\boxed{\texttt{r\#}}$ aims at allowing usage of Rust keywords as variable names; it is an escape sequence.

$$\underline{\text{\_variableDeclaration}} = \left\Downarrow \begin{array}{ll} \{\boxed{\texttt{r\#}}\}, & \\ \text{name} & :\in \mathbb{S}, \\ \{\boxed{\texttt{:␣}}\}, & \\ \text{type} & :\in \underline{\text{\_type}}, \\ \{\boxed{\texttt{,}\hookleftarrow}\} & \end{array} \right\Downarrow$$

$$\underline{\text{\_type}} = \{\boxed{\texttt{i32}}, \boxed{\texttt{i16}}, \boxed{\texttt{i8}}, \boxed{\texttt{bool}}\} \cup \underline{\text{\_tabular}}$$

$$\underline{\text{\_tabular}} = \left\Downarrow \begin{array}{ll} \{\boxed{\texttt{[}}\}, & \\ \text{type} & :\in \underline{\text{\_type}}, \\ \{\boxed{\texttt{;␣}}\}, & \\ \text{size} & :\in \mathbb{Z}, \\ \{\boxed{\texttt{]}}\} & \end{array} \right\Downarrow$$

$$\underline{\text{\_instruction}} = \underline{\text{\_assignement}} \cup \underline{\text{\_if}} \cup \underline{\text{\_functionCall}} \cup \underline{\text{\_while}}$$

$$\underline{\text{\_while}} = \left\Downarrow \begin{array}{ll} \{\boxed{\texttt{while␣}}\}, & \\ \text{condition} & :\in \underline{\text{\_predicate}}, \\ \{\boxed{\texttt{␣\{}\hookleftarrow}\}, & \\ \text{instructions} & :\in \underline{\text{\_instruction}}^{\infty}, \\ \{\boxed{\texttt{\}}\hookleftarrow}\} & \end{array} \right\Downarrow$$

$$\underline{\text{\_functionCall}} = \left\Downarrow \begin{array}{ll} \text{inits} & :\in \underline{\text{\_functionCallInit}}^{\infty}, \\ \{\boxed{\texttt{self.}}\}, & \\ \text{instanceName} & :\in \mathbb{S}, \\ \{\boxed{\texttt{.}}\}, & \\ \text{functionName} & :\in \mathbb{S}, \\ \{\boxed{\texttt{(}}\}, & \\ \text{inputParameters} & :\in \underline{\text{\_functionCallInputParameter}}^{\infty}, \\ \text{outputParameters} & :\in \underline{\text{\_functionCallOutputParameter}}^{\infty}, \\ \{\boxed{\texttt{);}\hookleftarrow}\}, & \end{array} \right\Downarrow$$

Initialization for output parameters (because Rust doesn't allow usage of uninitialized output parameters).

$$\underline{\text{\_functionCallInit}} = \left\Downarrow \begin{array}{ll} \{\boxed{\texttt{self.}}\}, & \\ \text{instanceName} & :\in \mathbb{S}, \\ \{\boxed{\texttt{␣=␣Default::default();}\hookleftarrow}\}, & \end{array} \right\Downarrow$$

$$\underline{\text{\_functionCallInputParameter}} = \left\Downarrow \begin{array}{ll} \{\boxed{\texttt{\&}}\}, & \\ \text{expression} & :\in \underline{\text{\_expr}}, \\ \{\boxed{\texttt{,}}\}, & \end{array} \right\Downarrow$$

Output parameters are mutable borrows.

$$\underline{\text{\_functionCallOutputParameter}} = \left\Downarrow \begin{array}{l} \{\boxed{\texttt{\&mut}\textvisiblespace}\}, \\ \text{name} \ :\in \underline{\text{\_variable}}, \\ \{\boxed{\hookleftarrow}\}, \end{array} \right\Downarrow$$

$$\underline{\text{\_assignement}} = \left\Downarrow \begin{array}{lll} \text{variable} & :\in & \mathbb{S}, \\ \{\boxed{\textvisiblespace\texttt{=}\textvisiblespace}\}, & & \\ \text{expr} & :\in & \underline{\text{\_expr}}, \\ \{\boxed{\texttt{;}\hookleftarrow}\} & & \end{array} \right\Downarrow$$

$$\underline{\text{\_expr}} = \underline{\text{\_int}} \cup \underline{\text{\_bool}} \cup \underline{\text{\_variable}} \cup \underline{\text{\_binaryexpression}} \cup \underline{\text{\_array}}$$

$$\underline{\text{\_int}} = \left[\!\left[\text{value} \ :\in \mathbb{Z} \right]\!\right] = \underline{\text{Integer\_Literal}} = \underline{\text{Int}}$$

$$\underline{\text{\_bool}} = \{\boxed{\texttt{true}}, \boxed{\texttt{false}}\}$$

$$\underline{\text{\_variable}} = \underline{\text{\_localVariable}} \cup \underline{\text{\_globalVariable}} \cup \underline{\text{\_parameterVariable}}$$

$$\underline{\text{\_localVariable}} = \left\Downarrow \begin{array}{l} \{\boxed{\texttt{r\#}}\}, \\ \text{name} \ :\in \mathbb{S} \end{array} \right\Downarrow$$

$$\underline{\text{\_globalVariable}} = \left\Downarrow \begin{array}{l} \{\boxed{\texttt{self.r\#}}\}, \\ \text{name} \ :\in \mathbb{S} \end{array} \right\Downarrow$$

They are necessarily borrows and need to be dereferenced first.

$$\underline{\text{\_parameterVariable}} = \left\Downarrow \begin{array}{l} \{\boxed{\texttt{*r\#}}\}, \\ \text{name} \ :\in \mathbb{S} \end{array} \right\Downarrow$$

$$\underline{\text{\_binaryexpression}} = \underline{\text{\_classicbinaryexpression}} \cup \underline{\text{\_exponentiation}} \cup \underline{\text{\_tabularAccess}}$$

$$\underline{\text{\_classicbinaryexpression}} = \left\Downarrow \begin{array}{lll} \{\boxed{\texttt{(}}\}, & & \\ \text{left\_expr} & :\in & \underline{\text{\_expr}}, \\ \text{type} & :\in & \underline{\text{\_binaryexprop}}, \\ \text{right\_expr} & :\in & \underline{\text{\_expr}}, \\ \{\boxed{\texttt{)}}\} & & \end{array} \right\Downarrow$$

$$\underline{\text{\_binaryexprop}} = \{\boxed{\texttt{)}\textvisiblespace\texttt{+}\textvisiblespace\texttt{(}}, \boxed{\texttt{)}\textvisiblespace\texttt{-}\textvisiblespace\texttt{(}}, \boxed{\texttt{)}\textvisiblespace\texttt{*}\textvisiblespace\texttt{(}}, \boxed{\texttt{)}\textvisiblespace\texttt{/}\textvisiblespace\texttt{(}}, \boxed{\texttt{)}\textvisiblespace\texttt{\%}\textvisiblespace\texttt{(}}\}$$

$$\underline{\text{\_tabularAccess}} = \left\Downarrow \begin{array}{lll} \text{tabular} & :\in & \underline{\text{\_expr}}, \\ \{\boxed{\texttt{[}}\}, & & \\ \text{index} & :\in & \underline{\text{\_expr}}, \\ \{\boxed{\texttt{]}}\} & & \end{array} \right\Downarrow$$

The `tabular` is an _expr because is could be itself a tabular access (case of a multiple dimensons tabular).

$$\text{\_exponentiation} = \left\Downarrow \begin{array}{lll} \{\boxed{\texttt{(}}\}, & & \\ \text{left\_expr} & :\in & \text{\_expr}, \\ \{\boxed{\texttt{).pow((}}\}, & & \\ \text{right\_expr} & :\in & \text{\_expr}, \\ \{\boxed{\texttt{).try\_into().unwrap())}}\} & & \end{array} \right\Downarrow$$

$$\text{\_array} = \left[\!\!\left[\begin{array}{lll} \{\boxed{\texttt{[}}\}, & & \\ \text{values} & :\in & \text{\_arrayExpr}^{\infty}, \\ \{\boxed{\texttt{]}}\}, & & \end{array} \right]\!\!\right]$$

$$\text{\_arrayExpr} = \left[\!\!\left[\begin{array}{lll} \text{value} & :\in & \text{\_expr}, \\ \{\boxed{\texttt{,\textvisiblespace}}\}, & & \end{array} \right]\!\!\right]$$

$$\text{\_if} = \left\Downarrow \begin{array}{lll} \{\boxed{\texttt{if\textvisiblespace}}\}, & & \\ \text{predicate} & :\in & \text{\_predicate}, \\ \{\boxed{\texttt{\textvisiblespace\{}\hookleftarrow}\}, & & \\ \text{then\_instr} & :\in & \text{\_instruction}^{\infty}, \\ \{\boxed{\texttt{\}\textvisiblespace else\textvisiblespace\{}\hookleftarrow}\}, & & \\ \text{else\_instr} & :\in & \text{\_instruction}^{\infty}, \\ \{\boxed{\texttt{\}}}\} & & \end{array} \right\Downarrow$$

$$\text{\_predicate} = \text{\_binarypredicate} \cup \text{\_predicateaggregate} \cup \text{\_unarypredicate}$$

$$\text{\_binarypredicate} = \left\Downarrow \begin{array}{lll} \text{leftexpr} & :\in & \text{\_expr}, \\ \text{symbol} & :\in & \text{\_comparisonop}, \\ \text{rightexpr} & :\in & \text{\_expr}, \end{array} \right\Downarrow$$

$$\text{\_comparisonop} = \{\boxed{\texttt{<}}, \boxed{\texttt{<=}}, \boxed{\texttt{=}}, \boxed{\texttt{>=}}, \boxed{\texttt{>}}, \boxed{\texttt{!=}}\}$$

$$\text{\_predicateaggregate} = \left\Downarrow \begin{array}{lll} \{\boxed{\texttt{(}}\}, & & \\ \text{firstpredicate} & :\in & \text{\_predicate}, \\ \text{aggregates} & :\in & \text{\_nestedpredicateaggregate}^{\infty}, \\ \{\boxed{\texttt{)}}\} & & \end{array} \right\Downarrow$$

$$\text{\_nestedpredicateaggregate} = \left\Downarrow \begin{array}{lll} \{\boxed{\texttt{)\textvisiblespace}}\}, & & \\ \text{the\_symbol} & :\in & \text{\_predicateaggregateop}, \\ \{\boxed{\texttt{\textvisiblespace(}}\}, & & \\ \text{predicate} & :\in & \text{\_predicate} \end{array} \right\Downarrow$$

$$\text{\_predicateaggregateop} = \{\boxed{\texttt{\&\&}}, \boxed{\texttt{||}}\}$$

$$
\underline{\text{\_unarypredicate}} = \left\Downarrow \begin{array}{l}
\{\boxed{\texttt{!(}}\}, \\
\text{predicate} :\in \underline{\text{\_predicate}}, \\
\{\boxed{\texttt{)}}\},
\end{array} \right\Downarrow
$$

$$
\underline{\text{\_instance\_init}} = \left\Downarrow \begin{array}{l}
\{\boxed{\texttt{self.}}\}, \\
\text{instanceName}_1 :\in \mathbb{S}, \\
\left\{\boxed{\texttt{␣=␣Default::default();}\hookleftarrow}\right., \\
\left.\boxed{\texttt{self.}}\right\}, \\
\text{instanceName}_2 :\in \mathbb{S}, \\
\left.\boxed{\texttt{.initialisation();}\hookleftarrow}\right\},
\end{array} \right\Downarrow
$$

$$
\underline{\text{\_function}} = \left\Downarrow \begin{array}{ll}
\{\boxed{\hookleftarrow\texttt{fn␣}}\}, & \\
\text{name} & :\in \mathbb{S}, \\
\{\boxed{\texttt{(\&mut␣self}}\}, & \\
\text{inputParameters} & :\in \underline{\text{\_inputParameter}}^{\infty}, \\
\text{outputParameters} & :\in \underline{\text{\_outputParameter}}^{\infty}, \\
\{\boxed{\texttt{)␣\{}\hookleftarrow}\}, & \\
\text{instructions} & :\in \underline{\text{\_instruction}}^{\infty}, \\
\{\boxed{\texttt{\}}\hookleftarrow}\} &
\end{array} \right\Downarrow
$$

$$
\underline{\text{\_inputParameter}} = \left\Downarrow \begin{array}{l}
\{\boxed{\texttt{,␣}}\}, \\
\text{name} :\in \mathbb{S}, \\
\{\boxed{\texttt{:␣\&}}\}, \\
\text{type} :\in \underline{\text{\_type}}
\end{array} \right\Downarrow
$$

It is an immutable reference.

$$
\underline{\text{\_outputParameter}} = \left\Downarrow \begin{array}{l}
\{\boxed{\texttt{,␣}}\}, \\
\text{name} :\in \mathbb{S}, \\
\{\boxed{\texttt{:␣\&mut␣}}\}, \\
\text{type} :\in \underline{\text{\_type}}
\end{array} \right\Downarrow
$$

6.6 From BXML to a converted shape

`b2rust` translates each `Parser` into a module. In this section, we will give this translation's scheme. A `Parser` can be seen as a couple of files: an abstract machine and an implementation. They can be seen as $a, i \in \mathcal{S}_{\text{BXML}}$ and the result of this translation, as $c \in \mathcal{S}_{\text{converted}}$.

6.6.1 Definitions

If m is a characters string, we refer by $P_a(m) \in \mathcal{S}_{\text{BXML}}$ the parsed shape of an abstract machine of a B module whose name is m (so, found in the file `name.bxml` if `name` is the name of the module). We refer by $P_i(m) \in \mathcal{S}_{\text{BXML}}$ the parsed shape of the implementation.

Concerning the imports of a module, the name of the instance written in the Rust code is not the one provided in the B code, because:

1. An import might not be renamed.
2. A concrete variable, for example, may have the same name as an "instance" (i.e. an import rename).

So, if our concrete variables and constants keep the same name, we need to add a prefix to our imports. We chose it to be $\boxed{\texttt{\_i\_}}$, where i is a counter reset for each module; if there's no instance name given, the name of the module is taken. If k is an natural number, the function ν_k give a `b2rust` instance name, given the B instance name and the B module name.

$$\nu_k : \begin{cases} \mathbb{S} \times \mathbb{S} & \longrightarrow \quad \mathbb{S} \\ (\boxed{\ }, n) & \longmapsto \quad \boxed{\texttt{\_}} \cdot k \cdot \boxed{\texttt{\_}} \cdot n \\ (r, n) & \longmapsto \quad \boxed{\texttt{\_}} \cdot k \cdot \boxed{\texttt{\_}} \cdot r \end{cases}$$

Remember the grammar RecognizedType; we need to represent it as a subset of $\mathcal{S}_{\text{BXML}}$ to use it in our translation scheme. So, RecognizedType is the set of expressions which are legit types you can give to your variables.

$$\mathsf{RecognizedType} = \left\{ \begin{array}{l} \left\{ i \mid i \in \mathsf{Id}, i.\mathsf{value} \in \left\{ \boxed{\texttt{rust\_i8}}, \boxed{\texttt{rust\_i16}}, \boxed{\texttt{rust\_i32}}, \boxed{\texttt{rust\_bool}} \right\} \right\} \cup \\ \left\{ b \mid b \in \mathsf{Binary\_Exp}, \left\{ \begin{array}{rl} b.\mathsf{exp}_1 & \in \left\{ n \mid n \in \mathsf{Binary\_Exp}, \left\{ \begin{array}{rl} n.\mathsf{exp}_1 & \in \mathsf{Integer\_Literal}, \\ n.\mathsf{exp}_1.\mathsf{value} & = 0, \\ n.\mathsf{exp}_2 & \in \mathsf{Integer\_Literal}, \\ n.\mathsf{op} & = \boxed{\texttt{..}} \end{array} \right\} \right\}, \\ b.\mathsf{exp}_2 & \in \mathsf{RecognizedType} \\ b.\mathsf{op} & = \boxed{\texttt{-->}}, \end{array} \right. \right\} \end{array} \right\}$$

T takes as argument a variable name and a pred_group and associates a string which is the type of the variable defined inside the predicate.

$$T : \begin{cases} \mathbb{S} \times \mathsf{pred\_group} & \longrightarrow \quad \mathsf{RecognizedType} \\ (v, p) & \longmapsto \quad \left\{ \begin{array}{l} \text{if we can define } e_c \in \mathsf{Exp\_Comparison} \text{ such as:} \\ \text{I)} \quad p \in \mathsf{Exp\_Comparison} \implies e_c = p, \\ \text{II)} \quad p \notin \mathsf{Exp\_Comparison} \implies e_c \Subset p.\mathsf{pred\_group}, \\ \text{III)} \quad e_c.\mathsf{Exp}_1 \in \mathsf{Id}, \\ \text{IV)} \quad e_c.\mathsf{Exp}_1.\mathsf{value} = s, \\ \text{V)} \quad e_c.\mathsf{Exp}_2 \in \mathsf{RecognizedType}, \\ \text{VI)} \quad e_c.\mathsf{op} = \boxed{\texttt{:}}, \\ \text{then, } e_c.\mathsf{Exp}_2.\mathsf{value}. \\ \text{Else, an unspecified value.} \end{array} \right. \end{cases}$$

The function τ_T, defined below, translate a BXML RecognizedType into a converted-shaped Type. According to the B specification, if the total function domain is an interval from 0 to a where $a \leq -1$, the array is empty.

$$\tau_T: \left\{ \begin{array}{lcl} \underline{\textsf{RecognizedType}} & \longrightarrow & \underline{\textsf{Type}} \\ \left[\,\boxed{\texttt{rust\_i8}}\,\right] & \longmapsto & \boxed{\textsf{i8\_t}} \\ \left[\,\boxed{\texttt{rust\_i16}}\,\right] & \longmapsto & \boxed{\textsf{i16\_t}} \\ \left[\,\boxed{\texttt{rust\_i32}}\,\right] & \longmapsto & \boxed{\textsf{i32\_t}} \\ \left[\,\boxed{\texttt{rust\_bool}}\,\right] & \longmapsto & \boxed{\textsf{bool\_t}} \\ \downarrow (\downarrow (\cdot, e_2'), \boxed{\texttt{..}} \downarrow, e_2), \boxed{\texttt{-->}} \downarrow \text{ if } e_2'.\textsf{value} \geq -1 & \longmapsto & \begin{bmatrix} \textsf{size} & : e_2'.\textsf{value} + 1, \\ \textsf{elementsType} & : \tau_T(e_2) \end{bmatrix} \\ \downarrow (\downarrow (\cdot, \cdot), \boxed{\texttt{..}} \downarrow, e_2), \boxed{\texttt{-->}} \downarrow \text{ elsewise} & \longmapsto & \begin{bmatrix} \textsf{size} & : 0, \\ \textsf{elementsType} & : \tau_T(e_2) \end{bmatrix} \end{array} \right.$$

The functions τ_{bool}, τ_{binexp}, τ_{id}, τ_{naryexp}, τ_{exp}, defined below, translate a boolean, a binary expression, a variable, an expression with multiple members, and, globally, an expression, respectively. The translation functions for integer and variables are not defined, because mathematically, they refer to the same objects. Note that τ_{exp} (and several other translation functions such as the one for predicates and instructions) also take a context as argument. This is needed by the τ_{id} function which needs to know the parameters of the operation it is in, so, it needs to be carried out for the functions which may nest an Id.

$$\tau_{\text{bool}}: \left\{ \begin{array}{ccl} \underline{\textsf{Boolean\_Literal}} & \longrightarrow & \mathbb{B} \\ b & \longmapsto & b.\textsf{value} \end{array} \right.$$

$$\tau_{\text{binexp}}: \left\{ \begin{array}{ccl} \underline{\textsf{Binary\_Exp}} \times \underline{\textsf{Context}} & \longrightarrow & \underline{\textsf{RustBinaryExpression}} \\ (\downarrow (e_1, e_2), o \downarrow, t) & \longmapsto & \begin{bmatrix} \textsf{left\_expr} & : \tau_{\text{exp}}(e_1, t), \\ \textsf{type} & : \tau_{\text{binexpop}}(o), \\ \textsf{right\_expr} & : \tau_{\text{exp}}(e_2, t) \end{bmatrix} \end{array} \right.$$

The function τ_{binexp} uses a translation for the symbols:

$$\tau_{\text{binexpop}}: \left\{ \begin{array}{ccl} \underline{\textsf{binary\_exp\_op}} & \longrightarrow & \underline{\textsf{binaryExpression}} \\ \boxed{\texttt{+i}} & \longmapsto & \boxed{\textsf{addition}} \\ \boxed{\texttt{-i}} & \longmapsto & \boxed{\textsf{substraction}} \\ \boxed{\texttt{*i}} & \longmapsto & \boxed{\textsf{multiplication}} \\ \boxed{\texttt{/i}} & \longmapsto & \boxed{\textsf{division}} \\ \boxed{\texttt{mod}} & \longmapsto & \boxed{\textsf{modulo}} \\ \boxed{\texttt{**i}} & \longmapsto & \boxed{\textsf{exponentiation}} \\ \boxed{\texttt{(}} & \longmapsto & \boxed{\textsf{tabularPosition}} \\ \cdot & \longmapsto & \cdot \end{array} \right.$$

$$\tau_{\text{naryexp}}: \left\{ \begin{array}{ccl} \underline{\textsf{Nary\_Exp}} \times \underline{\textsf{Context}} & \longrightarrow & \underline{\textsf{RustArray}} \\ \left(\left\downarrow \left(\left\downarrow (e_1^1, e_2^1), \right\downarrow, \ldots, \left\downarrow (e_1^n, e_2^n), \right\downarrow \right), \right\downarrow, t \right) & \longmapsto & \left\{ \begin{array}{l} \text{if we define:} \\ f: \left\{ \begin{array}{ccc} \mathbb{N} & \longrightarrow & \underline{\textsf{Exp}} \\ e_1^\kappa & \longmapsto & e_2^\kappa \end{array} \right\}, \text{ then,} \\ \downarrow (\tau_{\text{exp}}(f(0), t), \cdots, \tau_{\text{exp}}(f(n), t)) \downarrow \end{array} \right. \end{array} \right.$$

According to the specification of the BXML files, a Nary_Exp is always expected to be a relation definition (op $=$ $\boxed{\{}$); its inner Exp's are assumed to be Binary_Exp whose op is $\boxed{\texttt{|->}}$; the left members are assumed to be Integer_Literals; for any small enough integer k, the Nary_Exp defines a maplet associated to k (if $k = 6$, for example, the B code associated to this Nary_Exp may contain `6 |-> 67 + ii`). So, we can define a function (called f in the equation above) which associates the contained expression to every small enough integer.

$$\tau_{\text{id}}: \begin{cases} \underline{\mathsf{Id}} \times \underline{\mathsf{Context}} & \longrightarrow \underline{\mathsf{Variable}} \\ (d,t) \text{ if } d.\mathsf{value} \in t.\mathsf{global} & \longmapsto \left[\, d.\mathsf{value} \,\right] \in \underline{\mathsf{GlobalVariable}} \\ (d,t) \text{ if } d.\mathsf{value} \in t.\mathsf{parameters} & \longmapsto \left[\, d.\mathsf{value} \,\right] \in \underline{\mathsf{ParameterVariable}} \\ (d,t) \text{ otherwise} & \longmapsto \left[\, d.\mathsf{value} \,\right] \in \underline{\mathsf{LocalVariable}} \end{cases}$$

We need to have more information on the variable (and this is the reason why carring the context variable is useful) to translate it. This is because if the variable is a global variable or constant, it is translated as beeing a part of the `struct` and therefore needs to have a leading `self.`; if it is an operation parameter, it is a reference and we need to dereference it using a star.

$$\tau_{\text{exp}}: \begin{cases} \underline{\mathsf{Exp}} \times \underline{\mathsf{Context}} & \longrightarrow \underline{\mathsf{RustExpression}} \\ (e \in \underline{\mathsf{Integer\_Literal}}, \cdot) & \longmapsto e \\ (e \in \underline{\mathsf{Boolean\_Literal}}, \cdot) & \longmapsto \tau_{\text{bool}}(e) \\ (e \in \underline{\mathsf{Id}}, t) & \longmapsto \tau_{\text{id}}(e,t) \\ (e \in \underline{\mathsf{Binary\_Exp}}, t) & \longmapsto \tau_{\text{binexp}}(e,t) \\ (e \in \underline{\mathsf{Nary\_Exp}}, t) & \longmapsto \tau_{\text{naryexp}}(e,t) \\ \cdot & \longmapsto \bot \end{cases}$$

Then, we can define the function τ_{val} which translates a valuation into an instruction:

$$\tau_{\text{val}}: \begin{cases} \underline{\mathsf{Valuation}} \times \underline{\mathsf{Context}} & \longrightarrow \underline{\mathsf{RustAssignement}} \\ \left(\begin{bmatrix} \mathsf{ident} & : s, \\ \mathsf{Exp} & : (e_1, \cdots, e_n) \end{bmatrix}, t \right) & \longmapsto \begin{bmatrix} \mathsf{variable} & : \tau_{\text{id}}(s,t), \\ \mathsf{expression} & : \tau_{\text{exp}}(e_1,t) \end{bmatrix} \end{cases}$$

We also define the function τ_{op} which translates an operation:

$$
\tau_{\text{op}}: \begin{cases} \underline{\textsf{Operation}} \times \underline{\textsf{Context}} \longrightarrow \underline{\textsf{Function}} \\ (\downarrow o, u, r, b, n \downarrow, e) \longmapsto \begin{cases} \text{if we define:} \\ r' \in a.\textsf{Operations} \text{ such as} \\ \qquad\qquad\qquad r'.\textsf{name} = n, \\ e' \in \underline{\textsf{Context}} \quad \text{such as} \\ \qquad \begin{cases} e'.\textsf{global} & = e.\textsf{global}, \\ e'.\textsf{parameters} & = \{g \mid \exists d \Subset o \cup u, g = d.\textsf{value}\} \end{cases} \\ \text{then,} \\ \downarrow \begin{array}{ll} \textsf{name} & : n, \\ \textsf{inputParameters} & : \{(s,t) \mid \exists d \Subset o, s = d.\textsf{value}, t = \tau_T(T(s, r'))\}, \\ \textsf{outputParameters} & : \{(s,t) \mid \exists d \Subset u, s = d.\textsf{value}, t = \tau_T(T(s, r'))\}, \\ \textsf{instructions} & : \tau_{\text{inst}}((), b, e') \end{array} \downarrow \end{cases} \end{cases}
$$

For the sake of clarity, let us detail this using natural language.

- If we want to convert the input/output parameters, we need to know their types. The user needs to do it in the `Precondition` part of the operation, which is forbidden in the implementation. The function τ_{op} is used to translate implemented operations, of course, so, we need to search for the precondition of the abstraction of the operation (it is r'). Note that **the operation abstraction is identified by its name, because two operation cannot have the same name in B** (unless one is a rafinement of the other).
- e' is the context. It contains the global variables τ_{op} takes as argument, but it defines the list of parameters, so, the variables names are searched in $o \cup u$.
- Some "BXML instructions" (such as Nary_Sub, for example) can be translated into multiple "Rust instructions". As benefit, a lot of BXML instructions are merged into Rust instructions, so, there are much less Rust instructions than BXML instructions. The function τ_{inst}, defined below, takes as argument a tuple of RustInstruction and an instruction; it returns a completed tuple of RustInstruction. In the function τ_{op}, τ_{inst} is called with an empty tuple of RustInstruction.

$$
\tau_{\text{inst}}:\left\{\begin{array}{lcl}
\underline{\mathsf{RustInstruction}}^{\infty} \times \underline{\mathsf{Sub}} \times \underline{\mathsf{Context}} & \longrightarrow & \underline{\mathsf{RustInstruction}}^{\infty} \\
(i,s,o),\ \ s \in \underline{\mathsf{Assert\_Sub}} & \longmapsto & \tau_{\text{assert}}(i,s,o) \\
(i,s,o),\ \ s \in \underline{\mathsf{Assignement\_Sub}} & \longmapsto & \tau_{\text{assignment}}(i,s,o) \\
(i,s,o),\ \ s \in \underline{\mathsf{Becomes\_In}} & \longmapsto & \tau_{\text{becomesin}}(i,s,o) \\
(i,s,o),\ \ s \in \underline{\mathsf{If\_Sub}} & \longmapsto & \tau_{\text{if}}(i,s,o) \\
(i,s,o),\ \ s \in \underline{\mathsf{Nary\_Sub}} & \longmapsto & \tau_{\text{narysub}}(i,s,o) \\
(i,s,o),\ \ s \in \underline{\mathsf{Operation\_Call}} & \longmapsto & \tau_{\text{operationcall}}(i,s,o) \\
(i,s,o),\ \ s = \boxed{\mathsf{Skip}} & \longmapsto & i \\
(i,s,o),\ \ s \in \underline{\mathsf{VAR\_IN}} & \longmapsto & \tau_{\text{varin}}(i,s,o) \\
(i,s,o),\ \ s \in \underline{\mathsf{While}} & \longmapsto & \tau_{\text{while}}(i,s,o) \\
\cdot & \longmapsto & \cdot
\end{array}\right.
$$

With the following definitions:

$$
\tau_{\text{operationcall}}:\left\{\begin{array}{lcl}
\underline{\mathsf{RustInstruction}}^{\infty} \times \underline{\mathsf{Operation\_Call}} \times \underline{\mathsf{Context}} & \longrightarrow & \underline{\mathsf{RustInstruction}}^{\infty} \\
((i_1,\cdots,i_n),s,o)\ \ \text{if } s.\mathsf{name.id.instance} = \text{❏} & \longmapsto & \left\{\begin{array}{l}
\text{if we define:} \\
v = s.\mathsf{name.id.value}, \\
(v,t) \text{ such as } (v,t) \in o.\mathsf{operationsNameAssoc}, \\
\text{then:} \\
\left(i_1,\cdots,i_n,\begin{bmatrix} \mathsf{moduleName} & : t, \\ \mathsf{functionName} & : v, \\ \mathsf{inputParameters} & : \lambda_{i,s}, \\ \mathsf{inputParameters} & : \lambda_{o,s}, \end{bmatrix}\right)
\end{array}\right. \\
((i_1,\cdots,i_n),s,o)\ \ \text{elsewise} & \longmapsto & \left\{\begin{array}{l}
\text{if we define:} \\
t = s.\mathsf{name.id.instance}, \\
(t,\alpha) \text{ such as } (t,\alpha) \in o.\mathsf{instancesNameAssoc}, \\
\text{then:} \\
\left(i_1,\cdots,i_n,\begin{bmatrix} \mathsf{moduleName} & : \alpha, \\ \mathsf{functionName} & : s.\mathsf{name.id.component}, \\ \mathsf{inputParameters} & : \lambda_{i,s}, \\ \mathsf{inputParameters} & : \lambda_{o,s}, \end{bmatrix}\right)
\end{array}\right.
\end{array}\right.
$$

where:

$$\lambda_{i,s} = (\tau_{\text{exp}}(s.\mathsf{inputParameters.exp}_1),\cdots,\tau_{\text{exp}}(s.\mathsf{inputParameters.exp}_n))$$
$$\lambda_{o,s} = (\tau_{\text{exp}}(s.\mathsf{outputParameters.exp}_1),\cdots,\tau_{\text{exp}}(s.\mathsf{outputParameters.exp}_n))$$

Explanations:

- If the complete operation name (s.name.id.value) is precedated by a renamed instance (second case), we just need to find the name of the instance `b2rust` associated. Therefore, we use the object `instancesNameAssoc` in the context. The operation name is called `component` is the BXML file.
- Elsewise, i.e. if no instance name is given, it is more difficult as we need to "guess" the instance the operation is in (the BXML contains not enough

information). We use the `operationsNameAssoc` which is completed elsewhere.

$$\tau_{\text{narysub}}: \begin{cases} \underline{\mathsf{RustInstruction}}^{\infty} \times \underline{\mathsf{Nary\_Sub}} \times \underline{\mathsf{Context}} & \longrightarrow \underline{\mathsf{RustInstruction}}^{\infty} \\ (l, s, o) & \longmapsto \tau_{\text{inst}}(\cdots \tau_{\text{inst}}(\tau_{\text{inst}}(l, s.\mathsf{Sub}_1, o), s.\mathsf{Sub}_2, o) \cdots, s.\mathsf{Sub}_n, o) \end{cases}$$

$$\tau_{\text{assert}}: \begin{cases} \underline{\mathsf{RustInstruction}}^{\infty} \times \underline{\mathsf{Assert\_Sub}} \times \underline{\mathsf{Context}} & \longrightarrow \underline{\mathsf{RustInstruction}}^{\infty} \\ (l, \downarrow g, b \downarrow, o) & \longmapsto \tau_{\text{inst}}(l, b, o) \end{cases}$$

It just ignores the guard.

$$\tau_{\text{assignment}}: \begin{cases} \underline{\mathsf{RustInstruction}}^{\infty} \times \underline{\mathsf{Assignement\_Sub}} \times \underline{\mathsf{Context}} & \longrightarrow \underline{\mathsf{RustInstruction}}^{\infty} \\ ((i_1, \cdots, i_n), b, o) & \longmapsto \left(i_1, \cdots, i_n, \begin{bmatrix} \mathsf{variable} & : \tau_{\text{exp}}(b.\mathsf{Variables}_1, o), \\ \mathsf{expression} & : \tau_{\text{exp}}(b.\mathsf{Values}_1, o) \end{bmatrix}\right) \end{cases}$$

$$\tau_{\text{if}}: \begin{cases} \underline{\mathsf{RustInstruction}}^{\infty} \times \underline{\mathsf{If\_Sub}} \times \underline{\mathsf{Context}} & \longrightarrow \underline{\mathsf{RustInstruction}}^{\infty} \\ ((i_1, \cdots, i_n), s, o) & \longmapsto \left(i_1, \cdots, i_n, \begin{bmatrix} \mathsf{predicate} & : \tau_{\text{pred}}(s.\mathsf{Condition}, o), \\ \mathsf{then\_instr} & : \tau_{\text{inst}}((), s.\mathsf{Then}, o), \\ \mathsf{else\_instr} & : \tau_{\text{inst}}((), s.\mathsf{Else}, o) \end{bmatrix}\right) \end{cases}$$

And also:

$$\tau_{\text{pred}}: \begin{cases} \underline{\mathsf{predicate\_type}} \times \underline{\mathsf{Context}} & \longrightarrow \underline{\mathsf{RustPredicate}} \\ (\downarrow (e_1, e_2), o \downarrow \in \underline{\mathsf{Exp\_Comparison}}, n) & \longmapsto \begin{bmatrix} \mathsf{left\_expr} & : \tau_{\text{exp}}(e_1, n), \\ \mathsf{symbol} & : \tau_{\text{comparisonexp}}(o), \\ \mathsf{right\_expr} & : \tau_{\text{exp}}(e_2, n) \end{bmatrix} \\ \left(\begin{bmatrix} \mathsf{pred\_group} & : (p_1, \cdots, p_n), \\ \mathsf{op} & : o \end{bmatrix} \in \underline{\mathsf{Nary\_Pred}}, n\right) & \longmapsto \begin{bmatrix} \mathsf{predicates} & : (\tau_{\text{pred}}(p_1, n), \cdots, \tau_{\text{pred}}(p_n, n)), \\ \mathsf{symbol} & : \tau_{\text{comparisonexp}}(o) \end{bmatrix} \\ (d \in \underline{\mathsf{Unary\_Pred}}, n) & \longmapsto \left[\, \tau_{\text{pred}}(d.\mathsf{pred\_group}, n) \,\right] \\ \cdot & \longmapsto \cdot \end{cases}$$

The translation function used is:

$$\tau_{\text{comparisonexp}}: \begin{cases} \underline{\mathsf{comparison\_op}} & \longrightarrow \underline{\mathsf{comparisonSymbol}} \\ \boxed{\texttt{<i}} & \longmapsto \boxed{\mathsf{LessThan}} \\ \boxed{\texttt{<=i}} & \longmapsto \boxed{\mathsf{LessThanOrEqualTo}} \\ \boxed{\texttt{=}} & \longmapsto \boxed{\mathsf{EqualTo}} \\ \boxed{\texttt{/=}} & \longmapsto \boxed{\mathsf{NotEqualTo}} \\ \boxed{\texttt{>=i}} & \longmapsto \boxed{\mathsf{MoreThanOrEqualTo}} \\ \boxed{\texttt{>i}} & \longmapsto \boxed{\mathsf{MoreThan}} \\ \boxed{\texttt{\&}} & \longmapsto \boxed{\mathsf{And}} \\ \boxed{\texttt{or}} & \longmapsto \boxed{\mathsf{Or}} \\ \cdot & \longmapsto \boxed{\mathsf{MoreThan}} \end{cases}$$

$$\tau_{\text{becomesin}}: \begin{cases} \underline{\mathsf{RustInstruction}}^{\infty} \times \underline{\mathsf{Becomes\_In}} \times \underline{\mathsf{Context}} & \longrightarrow \underline{\mathsf{RustInstruction}}^{\infty} \\ ((i_1, \cdots, i_n), s, \cdot) & \longmapsto \left(i_1, \cdots, i_n, \begin{bmatrix} \mathsf{name} & : s.\mathsf{Variables.\_Id}_1.\mathsf{value}, \\ \mathsf{type} & : \tau_T(s.\mathsf{Value.\_exp.value}) \end{bmatrix}\right) \end{cases}$$

A `Becomes_In` instruction is translated by a declaration of a new variable, so, for example, `temp1 :: rust_bool` would be translated by the Rust code `let temp1: bool;`. The checker (which is specified in the section 5) ensures a `Becomes_In` instruction shall never appear outside the very first elements of a `VAR_IN`, so, these instruction will be valid. This is also the reason why the context is useless: a declaration is nothing more than a declaration.

$$\tau_{\text{varin}}: \begin{cases} \underline{\mathsf{RustInstruction}}^{\infty} \times \underline{\mathsf{VAR\_IN}} \times \underline{\mathsf{Context}} & \longrightarrow \underline{\mathsf{RustInstruction}}^{\infty} \\ (l, s, o) & \longmapsto \tau_{\text{inst}}(l, s.\mathsf{Body}, o) \end{cases}$$

The translation is very simple because the local variables declarations are directly inside the `Body`: according to the recognized language, which the checker checks, if n local variables are defined, the n first instructions must be `Becomes_In`. These instruction will be translated like any other one.

$$\tau_{\text{while}}: \begin{cases} \underline{\mathsf{RustInstruction}}^{\infty} \times \underline{\mathsf{While}} \times \underline{\mathsf{Context}} & \longrightarrow \underline{\mathsf{RustInstruction}}^{\infty} \\ ((i_1, \cdots, i_n), \downarrow d, b, \cdot, \cdot \downarrow, o) & \longmapsto \left(i_1, \cdots, i_n, \begin{bmatrix} \mathsf{condition} & : \tau_{\text{pred}}(d), \\ \mathsf{instructions} & : \tau_{\text{inst}}(\{\}, b, o) \end{bmatrix}\right) \end{cases}$$

6.6.2 Properties verified by a, i and c

Let us define n_v the number of valuations: $n_v = |i.\mathsf{Values}|$.

Then, a, i and c verify the following properties:

1. The $c.\mathsf{mods}$ only concerns the entry module: it has to define every recursive subimport. So, if (a, i) is not the entry module, $c.\mathsf{mods} = \varnothing$, and if it is:

$$c.\mathsf{mods} = \left\{ s \mid \exists i_2, \cdots, i_n \in \mathcal{S}_{\text{BXML}}, \begin{cases} \begin{cases} \forall k \in \{2, \cdots, n-1\}, \\ \exists j \{1, \cdots, |i_k.\mathsf{Imports.Referenced\_Machine}|\}, \\ i_{k+1} = P_i(i_k.\mathsf{Imports.Referenced\_Machine}_j.\mathsf{Name}) \end{cases} \\ \begin{cases} \exists j \{1, \cdots, |i.\mathsf{Imports.Referenced\_Machine}|\}, \\ i_2 = P_i(i.\mathsf{Imports.Referenced\_Machine}_j.\mathsf{Name}) \end{cases} \\ i_n = P_i(s) \end{cases} \right\}$$

2. Contrary to $c.\mathsf{mods}$, the $c.\mathsf{uses}$ only concerns the modules which are not the entry module. It contains the imported modules (if the entry module imports modules, the `mod` keywords are sufficient). So, if (a, i) is the entry module, $c.\mathsf{uses} = \varnothing$, and if it is not:

$$c.\mathsf{uses} = \left\{ s \mid \exists j, i.\mathsf{Imports.Referenced\_Machine}_j.\mathsf{Name} \right\}$$

3. The name is translated:

$$c.\mathsf{name} = i.\mathsf{Abstraction}$$

4. $c.\mathsf{instances}$ contains the imported modules of the module, entry module or not.

(a) There's exactly the imports of the implementation: $n \coloneqq |c.\mathsf{instances}| = |i.\mathsf{Imports.Referenced\_Machine}|$.

(b) The instances are translated in order, and their name in the Rust code is renamed: $\forall k\{1, \cdots, n\}$, if we call $s_k = i.\mathsf{Imports.Referenced\_Machine}_k$, then, $c.\mathsf{instances}_k = (\nu_k(s_k.\mathsf{Instance}), s_k.\mathsf{Name})$.

5. Each concrete variable among the refinement chain is translated with its type:

$$\forall v \Subset a.\mathsf{Concrete\_Variables} \cup i.\mathsf{Concrete\_Variables}, (v.\mathsf{value}, \tau_T\,(T(v.\mathsf{value}, i.\mathsf{Invariant}))) \in c.\mathsf{variables}$$

6. Each concrete constant among the refinement chain is translated with its type:

$$\forall v \Subset a.\mathsf{Concrete\_Constants} \cup i.\mathsf{Concrete\_Constants}, (v.\mathsf{value}, \tau_T\,(T(v.\mathsf{value}, i.\mathsf{Properties}))) \in c.\mathsf{variables}$$

7. Conversely, each occurrence of a variable with its converted type in the converted object comes from a B concrete variable or B concrete constant with the same type:

$$\begin{aligned}\forall (v,t) \in c.\mathsf{variables},\ & v \Subset a.\mathsf{Concrete\_Variables} \cup i.\mathsf{Concrete\_Variables} \wedge \tau_T(T(v.\mathsf{value}, i.\mathsf{Invariant}) = t \vee \\ & v \Subset a.\mathsf{Concrete\_Constants} \cup i.\mathsf{Concrete\_Constants} \wedge \tau_T(T(v.\mathsf{value}, i.\mathsf{Properties})) = t\end{aligned}$$

The next properties concern the initialisation, values, and functions translation. Let us introduce the following context object:

$$e = \left[\begin{array}{ll}\mathsf{global} & : \{v \mid \exists k \in \{1 \cdots |c.\mathsf{variables}|\}, v = c.\mathsf{variables}_k\}, \\ \mathsf{parameters} & : \{\} \\ \mathsf{instancesNameAssoc} & : \left\{(t,r) \mid \exists k, \left\{\begin{array}{l} t\ = i.\mathsf{Imports.Referenced\_Machine}_k.\mathsf{Name}, \\ r\ = c.\mathsf{instances}_{k,1}\end{array}\right\}\right\}, \\ \mathsf{operationsNameAssoc} & : \left\{(o,n) \mid \left\{\begin{array}{l}\exists k, n = c.\mathsf{instances}_{k,1}, \\ \exists p \in P_i(i.\mathsf{Imports.Referenced\_Machine}_k.\mathsf{Name}).\mathsf{Operations}, \\ o = p.\mathsf{name}\end{array}\right.\right\}\end{array}\right]$$

which will be used in the next properties. `instancesNameAssoc` is a `map` which contains association between the instances name (like specified in the B code) and the instance name `b2rust` gives (they don't match). `operationsNameAssoc` is a `map` which contains any couple of association between an operation name and a `b2rust` given instance name which matches a B module imported by this implementation, and which wasn't renamed (no instance name given). This is useful for the operations call translation.

8. $c.\mathsf{instances\_init}$ contains the name of the instances of imported modules in the Rust code. It repeats the contents of $c.\mathsf{instances}$. (It might seem useless, but it helps the conversion from a `RustModule` to a printed code to be straightforward.) So, $n \coloneqq |c.\mathsf{instances\_init}| = |c.\mathsf{instances}|$, and, $\forall k \in \{1, \cdots, n\}$, $c.\mathsf{instances\_init}_k = c.\mathsf{instances}_{k,1}$.

According to the B specification, the order of initialization has importance.

9. The values field contains the concrete constants valuations, in order.

$$c.\mathsf{values} = (\tau_{\text{val}}(i.\mathsf{Values}_1, e), \cdots, \tau_{\text{val}}(i.\mathsf{Values}_{n_v}, e))$$

10. The converted initialisation contains the translation of the `INITIALISATION`.

$$c.\mathsf{initialisations} = \tau_{\text{inst}}(\{\}, i.\mathsf{Initialisation}, e)$$

11. Each operation in the implementation is translated by exactly one operation in the converted object, and each operation in the converted object can be associated with an unique operation in the implementation:

    (a) $\tau'_{\text{op}} : \begin{cases} i.\mathsf{Operations}\rangle & \longrightarrow c.\mathsf{functions}\rangle \\ o & \longmapsto \tau_{\text{op}}(o, e) \end{cases}$ (mind the codomain), and:

    (b) τ'_{op} is bijective.

6.7 From a converted shape to Rust code

After the conversion to a converted shape ($c \in \mathcal{S}_{\text{converted}}$), `b2rust` "prints" c, i.e. the Rust code, to a given file. Mathematically, it can be seen in this document as another conversion from c to a printed code $p \in \mathcal{S}_{\text{Rust}}$. Note that some details (added newlines, for example) are not worth specifying.

6.7.1 Definitions

The conversion from c to p is very straightforward (`b2rust` was designed in this purpose) and there's not much to say about it. So, we dare to satisfy ourselves of the mere declaration of ϕ_{function}, ϕ_{inst}, ϕ_{exp}, and ϕ_{var}, the functions which translate a function, an instruction, an expression and a variable declaration, respectively.

However, there are some nuances. Firstly, with ϕ'_{pred} function, restriction of the ϕ_{pred} function which prints a predicate to RustPredicateAggregate:

$$\phi'_{\text{pred}} : \begin{cases} \underline{\mathsf{RustPredicateAggregate}} \longrightarrow \underline{\mathsf{\_predicateaggregate}} \\ \begin{bmatrix} \mathsf{predicates} & : (p_1, \cdots, p_n), \\ \mathsf{symbol} & : o \end{bmatrix} \longmapsto \begin{cases} \text{if we define } \mu = \phi_{\text{predicateaggregateop}}(o) \text{, then:} \\ \left\downarrow \begin{array}{l} \boxed{(}, \\ \phi_{\text{pred}}(p_1), \\ \left(\begin{array}{c} \downarrow \boxed{)\sqcup}, \mu, \boxed{\sqcup(}, \phi_{\text{pred}}(p_2) \downarrow, \\ \cdots, \\ \downarrow \boxed{)\sqcup}, \mu, \boxed{\sqcup(}, \phi_{\text{pred}}(p_n) \downarrow, \end{array} \right), \\ \boxed{)} \end{array} \right\downarrow \end{cases} \end{cases}$$

With the following definition:

$$\phi_{\text{predicateaggregateop}}: \begin{cases} \underline{\textsf{comparisonSymbol}} & \longrightarrow \underline{\textsf{\_predicateaggregateop}} \\ \boxed{\textsf{And}} & \longmapsto \boxed{\texttt{\&\&}} \\ \boxed{\textsf{Or}} & \longmapsto \boxed{\texttt{||}} \\ \cdot & \longmapsto \cdot \end{cases}$$

Then, with the ϕ_{binexp} function which translates a binary expression:

$$\phi_{\text{binexp}}: \begin{cases} \underline{\textsf{RustBinaryExpression}} & \longrightarrow \underline{\textsf{\_binaryexpression}} \\ \begin{bmatrix} \textsf{left\_expr} & : e_1, \\ \boxed{\textsf{exponentiation}}, & \\ \textsf{right\_expr} & : e_2 \end{bmatrix} & \longmapsto \begin{array}{l} \boxed{\texttt{(}}, \\ \phi_{\text{exp}}(e_1), \\ \boxed{\texttt{).pow((}}, \\ \phi_{\text{exp}}(e_2), \\ \boxed{\texttt{).try\_into().unwrap())}} \end{array} \\ \begin{bmatrix} \textsf{left\_expr} & : e_1, \\ \boxed{\textsf{tabularPosition}}, & \\ \textsf{right\_expr} & : e_2 \end{bmatrix} & \longmapsto \begin{array}{l} \phi_{\text{exp}}(e_1), \\ \boxed{\texttt{[(}}, \\ \phi_{\text{exp}}(e_2), \\ \boxed{\texttt{)␣as␣usize]}} \end{array} \\ \begin{bmatrix} \textsf{left\_expr} & : e_1, \\ \textsf{type} & : t, \\ \textsf{right\_expr} & : e_2 \end{bmatrix} \text{ elsewise} & \longmapsto \begin{array}{l} \boxed{\texttt{(}}, \\ \phi_{\text{exp}}(e_1), \\ \phi_{\text{binexpop}}(t), \\ \phi_{\text{exp}}(e_2), \\ \boxed{\texttt{)}} \end{array} \end{cases}$$

With the following definition:

$$\phi_{\text{binexpop}}: \begin{cases} \underline{\textsf{binaryExpression}} & \longrightarrow \underline{\textsf{\_binaryexprop}} \\ \boxed{\textsf{addition}} & \longmapsto \boxed{\texttt{)␣+␣(}} \\ \boxed{\textsf{substraction}} & \longmapsto \boxed{\texttt{)␣-␣(}} \\ \boxed{\textsf{multiplication}} & \longmapsto \boxed{\texttt{)␣*␣(}} \\ \boxed{\textsf{division}} & \longmapsto \boxed{\texttt{)␣/␣(}} \\ \boxed{\textsf{modulo}} & \longmapsto \boxed{\texttt{)␣\%␣(}} \\ \cdot & \longmapsto \cdot \end{cases}$$

Two remarks might worth the attention. Firstly, the binary expression are translated with parentheses everywhere, to prevent precedence problems; as drawback, the generated code might be unreadable (we might end up with lines such as `self.value = (self.value) + (1);`). Secondly, the exponentiation uses an awful `try_into().unwrap()`. This is because the `pow` function require an unsigned integer as second argument, but the Rust compiler might interpret it as a signed integer, for instance in the expression `2.pow(self.value + 2)` if `value` is a `i32`. The `try_into()` method parses the argument, for example from `i32` to `u32`; the `unwrap()` method helps to call a panic if the `try_into` fails; but in our case, as the B verification of our programs allows us to ensure that the second members are always unsigned integers, we can just parse it using `try_into().unwrap()`. A similar problem happens with the tabularPosition translation; the position of an array needs to have the `usize` type, so, it has to be parsed.

The $\phi_{\text{functioncall}}$ function which translates a function call; its inner workings is clear, except for the field `inits`; so, $\forall c \in$ FunctionCall, if we define $n = |c.\mathsf{outputParameters}|$,

$$\phi_{\text{functioncall}}(c).\mathsf{inits} = \left(\downarrow \begin{array}{l} \boxed{\texttt{self.}}, \\ \phi_{\text{exp}}(c.\mathsf{outputParameters}_1), \\ \boxed{\texttt{␣=␣Default::default();}} \end{array} \downarrow, \cdots, \downarrow \begin{array}{l} \boxed{\texttt{self.}}, \\ \phi_{\text{exp}}(c.\mathsf{outputParameters}_n), \\ \boxed{\texttt{␣=␣Default::default();}} \end{array} \downarrow \right)$$

A Rust function call has to be precedated by as many initializations as output parameters. This is because Rust refuses to call a function on uninitialized parameters.

Then, some remarks written using sentences:

- A Declaration is translated using the shape `let tmp1: i32;`, for example.
- A Block is basically a new Rust context. It is translated by `{...}` (so, it just adds brackets). Remember that a local variables instruction in translated by a new block.
- The output parameters are translated by mutable references (all functions are prototypes and will modify the mutable parameters instead of returning anything), and the input parameters are translated by immutable references. Rust is a language which doesn't like references, and, to put it in a nutshell, we cannot have two variables which are mutable references to the same place in memory, and we cannot have one if we have more than zero immutable reference to it. However, as mutable references are associated to new separate object created just before the call, and disapear afterwards, it is good to go.
- Example of a RustArray translation: `[1, 2, 7, -2, 67 + 12]`. Example of a tabular translation: `[i32; 6]` for an array with six `i32` values.

6.7.2 Properties verified by c and p

c and p verify the following properties:

1. The mods, uses, and name of the module is translated:

$$\begin{aligned} p.\mathsf{mods} &= (\downarrow c.\mathsf{mods}_1 \downarrow, \cdots) \\ p.\mathsf{uses} &= (\downarrow c.\mathsf{uses}_1 \downarrow, \cdots) \\ p.\mathsf{name}_1 = p.\mathsf{name}_2 &= c.\mathsf{name} \end{aligned}$$

2. The instances are also translated, with repetitions:

$$p.\mathsf{instances} = \left(\downarrow c.\mathsf{instances}_1, c.\mathsf{instances}_2, c.\mathsf{instances}_2 \downarrow, \cdots\right)$$

3. Each variables declaration are translated into exactly one variable declaration, and conversely, each variable declaration in the Rust code can be associated with exactly one in the converted structure. Note that the order is not necessarily preserved.

   (a) $\phi'_{\text{var}} : \begin{cases} c.\mathsf{variables} & \longrightarrow p.\mathsf{variables}\rangle \\ v & \longmapsto \phi_{\text{var}}(v) \end{cases}$ (mind the domain and codomain), and:

   (b) ϕ'_{var} is bijective.

4. Instances initialisations are also translated, with repetitions:

$$p.\mathsf{instances\_init} = \left(\downarrow c.\mathsf{instances\_init}_1, c.\mathsf{instances\_init}_2 \downarrow, \cdots\right)$$

5. Valuations are translated in order and the same way.

6. Initialisations instructions are translated in order. So, if $c.\mathsf{initialisations} = (i_1, \cdots, i_{n_c})$ and $p.\mathsf{initialisations} = (i'_1, \cdots, i'_{n_p})$,

   (a) $n_c = n_p$, and:

   (b) $\forall k \in \{1, \cdots, n_c\}, i'_k = \phi_{\text{inst}}(i_k)$.

7. Each function in the converted structure matches with exactly one function in the Rust code, and conversely. Note that the order is not necessarily preserved.

   (a) $\phi'_{\text{function}} : \begin{cases} c.\mathsf{functions}\rangle & \longrightarrow p.\mathsf{functions}\rangle \\ f & \longmapsto \phi_{\text{function}}(f) \end{cases}$ (mind the domain and codomain), and:

   (b) ϕ'_{function} is bijective.

6.8 An example

The following example of B code is taken:

```
MACHINE
    counter
VARIABLES
    value, overflow
INVARIANT
    value: INT & overflow: BOOL &
    0 <= value & value <= MAXINT &
    ((overflow = TRUE) => (value = MAXINT))
INITIALISATION
    value := 0 || overflow := FALSE
OPERATIONS
    zero =
    BEGIN
        value := 0 || overflow := FALSE
    END;
    inc =
    PRE
        value <= MAXINT
    THEN
        IF value < MAXINT THEN
            value := value + 1
        ELSE
            overflow := TRUE
        END
    END
END
```

```
IMPLEMENTATION
    counter_i
REFINES
    counter
SEES b2rust_types
CONCRETE_VARIABLES
    value, overflow
INVARIANT
    value: rust_i32 &
    overflow: rust_bool
INITIALISATION
    value := 0;
    overflow := FALSE
OPERATIONS
    zero =
    BEGIN
        value := 0;
        overflow := FALSE
    END;
    inc =
    BEGIN
        IF value < 2147483647 THEN
            value := value + 1
        ELSE
            overflow := TRUE
        END
    END
END
```

This code is shown for the understanding of the reader; we shall not manipulate it directly in the translation procedure described in the document. `b2rust` works only on BXML files, and mathematically, a BXML file is represented by an element of $\mathcal{S}_{\mathrm{BXML}}$. We call a the abstraction BXML structure and i the implementation BXML structure.

So, admit that we have:

$$a = \left| \begin{array}{ll} \mathsf{Abstraction} & : \text{❏}, \\ \mathsf{Imports} & : \varnothing, \\ \mathsf{Values} & : (), \\ \mathsf{Concrete\_Constants} & : (), \\ \mathsf{Concrete\_Variables} & : (), \\ \mathsf{Properties} & : \{\}, \\ \mathsf{Invariant} & : \cdots, \\ \mathsf{Initialisation} & : \cdots, \\ \mathsf{Operations} & : \cdots \end{array} \right|$$

$$
i = \left| \begin{array}{ll}
\mathsf{Abstraction} & : \boxed{\texttt{counter}}, \\
\mathsf{Imports} & : \varnothing, \\
\mathsf{Values} & : (), \\
\mathsf{Concrete\_Constants} & : (), \\
\mathsf{Concrete\_Variables} & : ([\boxed{\texttt{value}}], [\boxed{\texttt{overflow}}]), \\
\mathsf{Properties} & : \{\}, \\
\mathsf{Invariant} & : \left[\begin{array}{ll} \mathsf{pred\_group} & : \left(\begin{array}{c} \downarrow [\boxed{\texttt{value}}], [\boxed{\texttt{rust\_i32}}], \boxed{:} \downarrow, \\ \downarrow [\boxed{\texttt{overflow}}], [\boxed{\texttt{rust\_bool}}], \boxed{:} \downarrow \end{array}\right), \\ \mathsf{op} & : \boxed{\&} \end{array}\right], \\
\mathsf{Initialisation} & : \left[\begin{array}{ll} \mathsf{Sub} & : \left(\begin{array}{c} \downarrow ([\boxed{\texttt{value}}]), ([0]) \downarrow, \\ \downarrow ([\boxed{\texttt{overflow}}]), ([\bot]) \downarrow, \end{array}\right), \\ \mathsf{op} & : \boxed{;} \end{array}\right], \\
\mathsf{Operations} & : \left(\begin{array}{c} \left| \begin{array}{l} (), \\ (), \\ \varnothing, \\ \left[\begin{array}{ll} \mathsf{Sub} & : \left(\begin{array}{c} \downarrow ([\boxed{\texttt{value}}]), ([0]) \downarrow, \\ \downarrow ([\boxed{\texttt{overflow}}]), ([\bot]) \downarrow, \end{array}\right), \\ \mathsf{op} & : \boxed{;} \end{array}\right], \\ \boxed{\texttt{zero}} \end{array} \right|, \\
\left| \begin{array}{l} (), \\ (), \\ \varnothing, \\ \left[\begin{array}{ll} \mathsf{Sub} & : \left(\left[\begin{array}{ll} \mathsf{Condition} & : \downarrow [\boxed{\texttt{value}}], [2147483647], \boxed{<} \downarrow, \\ \mathsf{Then} & : \downarrow ([\boxed{\texttt{value}}]), (\downarrow [\boxed{\texttt{value}}], [1], \boxed{+} \downarrow) \downarrow, \\ \mathsf{Else} & : \downarrow ([\boxed{\texttt{overflow}}]), ([\top]) \downarrow, \\ \mathsf{elseif} & : \boxed{\texttt{no}} \end{array}\right]\right), \\ \mathsf{op} & : \boxed{;} \end{array}\right], \\ \boxed{\texttt{inc}} \end{array} \right|, \end{array} \right) \end{array} \right|
$$

`b2rust` works on these elements we represent as i and a and converts them into an element we represent as $c \in \mathcal{S}_{\text{converted}}$. How does it look like? To know it, let us go over the different properties as given in section 6.6.2:

1. There's no imports in this example. So, $c.\mathsf{mods} = \varnothing$.
2. For the same reasons, $c.\mathsf{uses} = \varnothing$.
3. As $i.\mathsf{Abstraction} = \boxed{\texttt{counter}}$, $c.\mathsf{name} = \boxed{\texttt{counter}}$. This one is easy.
4. For the same reasons, $c.\mathsf{imports} = \varnothing$.
5. The property becomes :

$$\forall v \in \{\downarrow \boxed{\texttt{value}} \downarrow, \downarrow \boxed{\texttt{overflow}} \downarrow\}, (v.\mathsf{value}, \tau_T(T_{v,i}(v))) \in c.\mathsf{variables}$$

.

$T_{v,i}(\downarrow \boxed{\texttt{value}} \downarrow)$ is $\boxed{\texttt{rust\_i32}}$. This is because, as $i.\mathsf{Invariant.pred\_group} \notin$ Exp_Comparison (it is a Nary_Pred), $e_c \in i.\mathsf{Invariant.pred\_group}$. Only one invariant matches the properties III to VI: $(\downarrow \boxed{\texttt{value}} \downarrow, \downarrow \boxed{\texttt{rust\_i32}} \downarrow, \boxed{:})$.

For the same reasons, $T_{v,i}(\downarrow \boxed{\texttt{overflow}} \downarrow)$ is $\boxed{\texttt{rust\_bool}}$.

If we apply the τ_T type-conversion function, we get that

$$\{(\boxed{\texttt{value}}, \boxed{\mathsf{i32\_t}}), (\boxed{\texttt{overflow}}, \boxed{\mathsf{bool\_t}})\} \subset c.\mathsf{variables}.$$

6. We could follow the same rule, but as there's no concrete constant defined, it is useless. We only get that $\{\} \subset c.\mathsf{variables}$, which is not very useful.

7. This property ensures that

$$c.\mathsf{variables} = \{(\boxed{\texttt{value}}, \boxed{\mathsf{i32\_t}}), (\boxed{\texttt{overflow}}, \boxed{\mathsf{bool\_t}})\}.$$

   Let $(l, t) \in c.\mathsf{variables}$. The property ensures that l is defined somewhere as a concrete variable or concrete constant in a or i, so:

   (a) Either $l = \boxed{\texttt{value}}$. Then, $\exists v \in \mathsf{Id}, v.\mathsf{value} = \boxed{\texttt{value}}$ and in this case $(v, t) = (\downarrow \boxed{\texttt{value}} \downarrow, T_{v,i}(\downarrow \boxed{\texttt{value}} \downarrow)) = (\boxed{\texttt{value}}, \boxed{\mathsf{i32\_t}})$.

   (b) Or, $l = \boxed{\texttt{overflow}}$. Then, for the same reasons, we get that $(v, t) = (\boxed{\texttt{overflow}}, \boxed{\mathsf{bool\_t}})$.

8. There's no imports in this example. So, $c.\mathsf{instances\_init} = \varnothing$.

9. There is nothing is $i.\mathsf{Values}$, so, the equation becomes:

$$\begin{aligned}
c.\mathsf{initialisations} &= \tau_{\text{inst}}((), i.\mathsf{Initialisation}, \{\}) \\
&= \tau_{\text{narysub}}((), i.\mathsf{Initialisation}, \{\}) \\
&= \tau_{\text{inst}}(\tau_{\text{inst}}((), i.\mathsf{Initialisation.Sub}_1, \{\}), i.\mathsf{Initialisation.Sub}_2, \{\}) \\
&= \tau_{\text{inst}}\left(\tau_{\text{inst}}\left((), \downarrow \begin{matrix}([\boxed{\texttt{value}}]),\\ ([0])\end{matrix} \downarrow, \{\}\right), \downarrow \begin{matrix}([\boxed{\texttt{overflow}}]),\\ ([\bot])\end{matrix} \downarrow, \{\}\right) \\
&= \left(\begin{bmatrix} \mathsf{variable} & : \tau_{\text{exp}}(\boxed{\texttt{value}}, \{\}) \\ \mathsf{expression} & : \tau_{\text{exp}}(0, \{\}) \end{bmatrix}, \begin{bmatrix} \mathsf{variable} & : \tau_{\text{exp}}(\boxed{\texttt{overflow}}, \{\}) \\ \mathsf{expression} & : \tau_{\text{exp}}(\bot, \{\}) \end{bmatrix}\right)
\end{aligned}$$

   etc.

10. It tells us exactly the contents of $c.\mathsf{functions}$. We need to follow the translation functions.

Now we know exactly how does c look like:

$$c = \begin{bmatrix}
\mathsf{mods} & : \varnothing, \\
\mathsf{uses} & : \varnothing, \\
\mathsf{name} & : \boxed{\texttt{counter}}, \\
\mathsf{instances} & : \varnothing, \\
\mathsf{variables} & : \{(\boxed{\texttt{value}}, \boxed{\mathsf{i32_t}}), (\boxed{\texttt{overflow}}, \boxed{\mathsf{bool_t}})\}, \\
\mathsf{instances_init} & : \varnothing, \\
\mathsf{initialisations} & : \left(\begin{bmatrix} \mathsf{variable} & : \boxed{\texttt{value}}, \\ \mathsf{expression} & : \left[\, 0 \,\right] \end{bmatrix}, \begin{bmatrix} \mathsf{variable} & : \boxed{\texttt{overflow}}, \\ \mathsf{expression} & : \left[\, \bot \,\right] \end{bmatrix} \right), \\
\mathsf{functions} & : \left(\begin{bmatrix} \mathsf{name} & : \boxed{\texttt{zero}}, \\ \mathsf{inputParameters} & : \{\}, \\ \mathsf{outputParameters} & : \{\}, \\ \mathsf{instructions} & : \left(\begin{bmatrix} \mathsf{variable} & : \boxed{\texttt{value}}, \\ \mathsf{expression} & : \left[\, 0 \,\right] \end{bmatrix}, \begin{bmatrix} \mathsf{variable} & : \boxed{\texttt{overflow}}, \\ \mathsf{expression} & : \left[\, \bot \,\right] \end{bmatrix} \right) \end{bmatrix}, \begin{bmatrix} \mathsf{name} & : \boxed{\texttt{inc}}, \\ \mathsf{inputParameters} & : \{\}, \\ \mathsf{outputParameters} & : \{\}, \\ \mathsf{instructions} & : \left(\begin{bmatrix} \mathsf{predicate} & : \begin{bmatrix} \mathsf{left_expr} & : \left[\, \boxed{\texttt{value}} \,\right], \\ \mathsf{symbol} & : \boxed{\mathsf{LessThan}}, \\ \mathsf{right_expr} & : \left[\, 2147483647 \,\right] \end{bmatrix}, \\ \mathsf{then_instr} & : \left(\begin{bmatrix} \mathsf{variable} & : \boxed{\texttt{value}}, \\ \mathsf{expression} & : \begin{bmatrix} \mathsf{left_expr} & : \left[\, \boxed{\texttt{value}} \,\right], \\ \mathsf{type} & : \left[\, \boxed{\mathsf{addition}} \,\right], \\ \mathsf{right_expr} & : \left[\, 1 \,\right] \end{bmatrix} \end{bmatrix} \right), \\ \mathsf{else_instr} & : \left(\begin{bmatrix} \mathsf{variable} & : \boxed{\texttt{overflow}}, \\ \mathsf{expression} & : \left[\, \top \,\right] \end{bmatrix} \right) \end{bmatrix} \right) \end{bmatrix} \right)
\end{bmatrix}$$

`b2rust` then prints the converted object. In this document, it is represented as a translation to a $p \in \mathcal{S}_{\text{Rust}}$ element. Its translation is straightforward and we have:

$$
p = \left\downarrow \begin{array}{l}
\boxed{\texttt{\#[derive(Default)]}\hookleftarrow} \\
\boxed{\texttt{pub\textvisiblespace struct\textvisiblespace}}, \boxed{\texttt{counter}}, \boxed{\texttt{\textvisiblespace\{}\hookleftarrow}, \\
\boxed{\texttt{//\textvisiblespace Instances\textvisiblespace of\textvisiblespace imported\textvisiblespace modules.}\hookleftarrow}, \\
\boxed{\texttt{//\textvisiblespace Concrete\textvisiblespace variables\textvisiblespace\&\textvisiblespace constants.}\hookleftarrow}, \\
\left(\begin{array}{c}
\downarrow \boxed{\texttt{r\#}}, \boxed{\texttt{value}}, \boxed{\texttt{:\textvisiblespace}}, \boxed{\texttt{i32}}, \boxed{\texttt{,}\hookleftarrow} \downarrow, \\
\downarrow \boxed{\texttt{r\#}}, \boxed{\texttt{overflow}}, \boxed{\texttt{:\textvisiblespace}}, \boxed{\texttt{bool}}, \boxed{\texttt{,}\hookleftarrow} \downarrow
\end{array} \right), \\
\boxed{\texttt{\}}\hookleftarrow} \\
\boxed{\hookleftarrow} \\
\boxed{\texttt{impl\textvisiblespace}}, \boxed{\texttt{counter}}, \boxed{\texttt{\textvisiblespace\{\textvisiblespace}\hookleftarrow} \\
\boxed{\texttt{pub\textvisiblespace fn\textvisiblespace initialisation(\&mut\textvisiblespace self)\textvisiblespace\{}\hookleftarrow}, \\
\boxed{\texttt{//\textvisiblespace Instances\textvisiblespace of\textvisiblespace imported\textvisiblespace modules\textvisiblespace initialization.}\hookleftarrow}, \\
\boxed{\hookleftarrow\texttt{//\textvisiblespace `INITIALISATION`\textvisiblespace clause.}\hookleftarrow}, \\
\left(\begin{array}{c}
\downarrow \left[\boxed{\texttt{r\#}}, \boxed{\texttt{value}} \right], \boxed{\texttt{\textvisiblespace=\textvisiblespace}}, \left[\boxed{\texttt{0}} \right], \boxed{\texttt{;}\hookleftarrow} \downarrow, \\
\downarrow \left[\boxed{\texttt{r\#}}, \boxed{\texttt{overflow}} \right], \boxed{\texttt{\textvisiblespace=\textvisiblespace}}, \left[\left[\boxed{\texttt{false}} \right] \right], \boxed{\texttt{;}\hookleftarrow} \downarrow
\end{array} \right), \\
\boxed{\texttt{\}}\hookleftarrow}, \\
\left(\begin{array}{c}
\left\downarrow \begin{array}{l}
\boxed{\hookleftarrow} \\
\boxed{\texttt{pub\textvisiblespace fn\textvisiblespace}}, \boxed{\texttt{zero}}, \boxed{\texttt{(\&mut\textvisiblespace self)\textvisiblespace\{}\hookleftarrow}, \\
\left(\begin{array}{c}
\downarrow \left[\boxed{\texttt{r\#}}, \boxed{\texttt{value}} \right], \boxed{\texttt{=}}, \left[0 \right], \boxed{\texttt{;}\hookleftarrow} \downarrow, \\
\downarrow \left[\boxed{\texttt{r\#}}, \boxed{\texttt{overflow}} \right], \boxed{\texttt{=}}, \left[\boxed{\texttt{false}} \right], \boxed{\texttt{;}\hookleftarrow} \downarrow,
\end{array} \right), \\
\boxed{\texttt{\}}\hookleftarrow}
\end{array} \right\downarrow, \\
\left\downarrow \begin{array}{l}
\boxed{\hookleftarrow} \\
\boxed{\texttt{pub\textvisiblespace fn\textvisiblespace}}, \boxed{\texttt{inc}}, \boxed{\texttt{(\&mut\textvisiblespace self)\textvisiblespace\{}\hookleftarrow}, \\
\left(\left\downarrow \begin{array}{l}
\boxed{\texttt{if\textvisiblespace}}, \\
\downarrow \boxed{\texttt{self.}}, \boxed{\texttt{r\#}}, \boxed{\texttt{value}}, \boxed{\texttt{\textvisiblespace<\textvisiblespace}}, \boxed{\texttt{2147483647}} \downarrow, \\
\boxed{\texttt{\textvisiblespace\{}\hookleftarrow}, \\
\left(\downarrow \boxed{\texttt{r\#}}, \left[\boxed{\texttt{value}} \right], \boxed{\texttt{\textvisiblespace=\textvisiblespace}}, \downarrow \boxed{\texttt{r\#}}, \left[\boxed{\texttt{value}} \right], \boxed{\texttt{+}}, \left[\boxed{\texttt{1}} \right] \downarrow, \boxed{\texttt{;}\hookleftarrow} \downarrow \right), \\
\boxed{\texttt{\}\textvisiblespace else\textvisiblespace\{}}, \\
\left(\downarrow \left[\boxed{\texttt{r\#}}, \boxed{\texttt{overflow}} \right], \boxed{\texttt{\textvisiblespace=\textvisiblespace}}, \left[\boxed{\texttt{true}} \right], \boxed{\texttt{;}\hookleftarrow} \downarrow \right), \\
\boxed{\texttt{\}}\hookleftarrow}
\end{array} \right\downarrow \right), \\
\boxed{\texttt{\}}\hookleftarrow}
\end{array} \right\downarrow
\end{array} \right), \\
\boxed{\texttt{\}}\hookleftarrow}
\end{array} \right\downarrow
$$

The output of `b2rust` will be the following:

```
#[derive(Default)]
pub struct counter {
// Concrete variables & constants.
r#overflow: bool,
r#value: i32,
}
```

```
impl counter {
pub fn initialisation(&mut self) {
// `INITIALISATION` clause.
self.r#value = 0;
self.r#overflow = false;
}

pub fn zero(&mut self) {
self.r#value = 0;
self.r#overflow = false;
}

pub fn inc(&mut self) {
if self.r#value < 2147483647 {
self.r#value = (self.r#value) + (1);
} else {
self.r#overflow = true;
}
}
}
```

The output can then be formatted using another program, for example `rustfmt`. There are some differences with the expecting behavior, concerning the newlines and comments. For a sake of simplicity, they have not been correctly specified.

7 Inner workings of `b2rust`

Although a B0 program is quite algorithmic and one can think its translation might be straightforward, the inner workings of `b2rust` can be interesting, for instance for the implementation of another converter, or, of course, for maintenance purposes.

`b2rust` is programmed with the following technologies, by decreasing order of importance:

- C++, for all the code concerning the execution of `b2rust`, the parsing of the BXML, its conversion, and the printing of the Rust code.
- Bash, for all the tests scripts.
- CMake, for the compilation.
- Rust and XML, for the test reference files.

7.1 The testing

All the files related to the testing can be found in the `tests` directory. If you want to test `b2rust`, the section 3.3 should be the one you are looking for. Here, we will only explain its technical aspects.

Each category of test uses a script, `test.sh`, you can find in its dedicated directory; however, this script is not invoked manually; CMake, invoked when testing with the `ctest .` command, reads a `tests_file.cmake` which contains the commands compelling it to run the scripts with correct parameters; however, the lines in this file are generated automatically so that a tester is not compelled to add a line manually. It is the purpose of the script `gen_tests.sh`, automatically called by the `cmake .` command, which goes over each category of test to find all its tests.

7.2 The compilation

There is not much to say; CMake uses the `CMakeLists.txt` instruction file which, in particular, orders CMake to run the `gen_tests.sh` script each time a user executes `cmake .` and to tell it what to test when running `ctest .`; at the beginning of the file, it has options which concern the compilation and can be changed.

7.3 Debugging

The important classes overload the C++ `<<` operator to help a developer to debug `b2rust`. It allows the developer just to write `std::cout << object` to print most useful objects.

8 Development conventions

8.1 In the conversion code

8.1.1 `const` and `private` whenever possible

Please use `const` whenever possible, even it this means writing things such as `const char* const* const` and writing public accessors. Its usage just needs to:

- Be able to compile (fortunately),
- Be useful, i.e. mean anything (no `new const Object`, for example).

It shall help having a bug-free code. However, this convention is not currently is use everywhere, because it is time-consuming.

8.1.2 `std::string`

If you contribute to the development of `b2rust`, please use `std::string` objects instead of `const char*` or `char*` ones. `tinyxml2` functions use `const char*` values as argument, so it is the purpose of the `tinyxml2ext` component to parse the arguments; but except for the `main` and `tinyxml2ext` components, the use of `std::string` shall be preferred.

If an use of `char*` is absolutely mandatory (e.g. because you need to use an external library), use the comment ”This one is fine.” on the same line as the `char` usage. This will allow us to check the absence of `char*`s in the whole code using a command line `grep -rn "char\*"`, for example.

8.1.3 Various good C++ practices

Respect recommended C++ practices, e.g. never use `using namespace std` and never use `using namespace` in header files.

8.2 In the Git repository

Please only push code which compiles, and, if possible, whose documentation is up-to-date.

9 Appendix

9.1 Error codes of `b2rust`

`OK`	0	The program behaved as expected. This does not mean no error occurred, but if an error occurred, it did not concern the conversion (i.e. the conversion would have been the same even without this error and is likely correct).
`ERR_SYNTAX`	1	The call syntax of the program was not respected.
`ERR_OUTPUT_FILE`	2	The output file could not be created, or could not be opened with the write permission.
`ERR_OUTPUT_STREAM`	3	Could not write to output stream. R/W or logical error on I/O operation.
`ERR_BXML_LOADING`	10	The BXML file could not be loaded.
`ERR_BXML_CHECKING`	20	An error occurred during the BXML checking. There’s at least one error. Either the file is not BXML compliant, or an element cannot be converted yet.
`ERR_BXML_LACK`	21	An error occurred during the conversion. The file is BXML compliant and all its elements can be converted, but an element is lacking to allow the conversion. Do your files represent a verified B program? If yes, it is a b2rust error.

9.2 `b2rust` relevant Rust types

Type name	Description
`i32`	A signed integer coded on 32 bits.
`i16`	A signed integer coded on 16 bits.
`i8`	A signed integer coded on 8 bits.
`bool`	A boolean.

9.3 Assumed BXML specification errors

The BXML 1.0 specification contains imprecisions which are obvious enough we suppose they will be corrected. The next assumptions might be made during the specification of `b2rust`:

1. Inside the `Operation_Call` element, inside the `Output_Parameters` element: a sequence of `Exp` is specified; we suppose it is a sequence of `Id`.

END OF DOCUMENT.