b2rust

User Manual

CLEARSY

2023-07-17

Introduction

This document serves as a **user manual** for b2rust. It presents its **constrains on use** and the **translation choices** of B models to Rust.

Run b2rust

Configuration

Resource file

b2rust is capable of translating the following constructions in implementation components :

— **vv : : E** (Becomes in)
— **vv : (vv : E)** (Becomes such that)

Configure the type checker to accept these constructions by adding the following lines to your project's **AtelierB** resource file :

```
ATB*BCOMP*Allow_Becomes_Member_Of:TRUE
ATB*BCOMP*Allow_Becomes_Such_That:TRUE
ATB*TC*Allow_Becomes_Member_Of:TRUE
ATB*TC*Allow_Becomes_Such_That:TRUE
```

Configuration files

b2rust needs to know the directory where its three configuration files are located :

-b2rust__types.cfg -b2rust__operations.cfg -b2rust__exceptions.cfg.

The directory may be given as a command line parameter with the option `-c`, or with environment variable `B2RUST_CONF_HOME`. The command line parameter has precedence over the environment variable.

Default configuration files are distributed with b2rust in the `files/` directory. The same directory contains templates in case you want to change the configuration for your needs.

Code generation

BXML generation

b2rust actually translates the 'bxml'. A script file named 'gen_bxml.sh' is provided to create the bxml from the **mch**, **ref**, and **imp** extension files.

To run this script, it is recommended to add the path to the bxml **executable** from AtelierB to the native library :

```
export LD_LIBRARY_PATH=/path/to/atelierB/bbin/linux_x64/:$LD_LIBRARY_PATH
```

Now, the **gen_bxml** script :

```
sh gen_bxml.sh $1 $2 $3
```

where

- **$1** : the path to the directory containing the bxml **executable** from AtelierB
- **$2** : the target directory containing the files for which you want to generate the 'bxml'
- **$3** : the AtelierB resource file

Génération du code Rust

After generating the bxml, b2rust is capable of translating B models.

To translate a B component, use the following command :

```
Usage:
  b2rust [-h | --help]
  b2rust [-v | --version]
  b2rust -i src [ -c cfgpath ] ( -l lib )* [ -o dst ] component
Options:
  -c, --configuration path  Sets the path to the configuration directory
  -I, -i, --include path    Sets the path to a directory containing BXML files of the main p
  -l, --library path        Sets the path to a directory containing BXML files of a library
  -o, --output path         Sets the path to a directory where the generated Rust files are
  -h, --help                Display this help message and exits
  -v, --version             Displays the program version and exits
```

This command also recursively translates all modules that are seen, imported, and extended by the module of the given component.

Example :

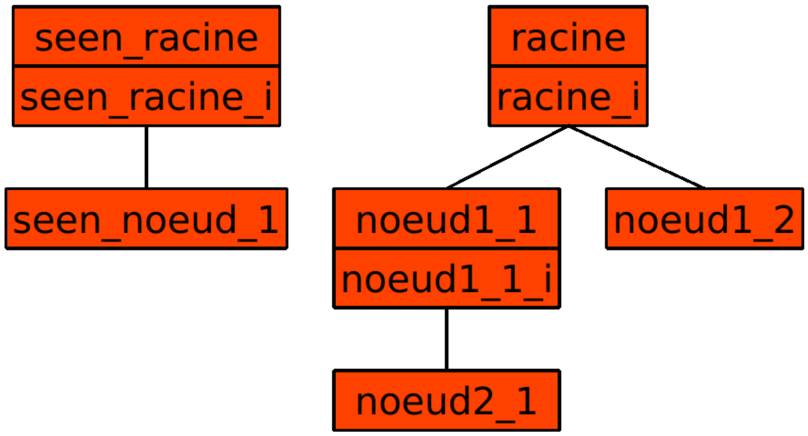

FIGURE 1 – Translation tree

— Applying b2rust to the machine “noeud1_1” translates the machines in its subtree “noeud1_1” and “noeud2_1”, but not the root machines “noeud1_2”.
— If “seen_racine” is seen by a machine in the subtree of “noeud1_1”, then the machines “seen_racine” and “seen_noeud” are also translated.

B0 in b2rust

b2rust translates only the **implementation** of machines and the **basic modules**. Generally, the translation pattern is :

— Each **machine** is translated into a rust **struct**.
— Concrete_variables and Referenced machines become struct fields.
— Concrete_constants become the struct's static constants.
— Machine operations become struct methods.

Typing

b2rust only translates concrete_constants and concrete_variables (concrete data). Each concrete data must be typed at least once in the module in order to be translated :

— **Constants** must be typed in the **PROPERTIES** clause.
— **Variables** must be typed in the **INVARIANT** clause.

Every concrete data that has 'ident' as identifiant must be typed using the BelongTo operator' **:**' :

ident : rust_type

b2rust determines the data type according to the rust_type **identifier**. To make things easy, all machines should **see** b2rust_types.mch, which already provides a good definition of each identifier.

Each concrete date cannot be typed several times within the same machine, but they can be typed **differently** between abstract machines and their refinements. Only the **latter** matters for b2rust.

Atomic type

Non terminal	Productions
rust_integer	: := rust_signed_integer | rust_unsigned_integer
rust_bool	: := TRUE | FALSE
rust_string	: := ‘“’ .* ‘“’

Non terminal	Productions
rust_signed_integer	: := rust_i8 |rust_i16 |rust_i32 |rust_i64 |rust_i128
rust_unsigned_integer	: := rust_u8 |rust_u16 |rust_u32 |rust_u64 |rust_u128

Note that in AtelierB, STRING type is exclusive to operation parameters. Click here to see an example of STRING type use

B code example :

```
MACHINE
    atomic_type

SEES
    b2rust_types

CONCRETE_CONSTANTS
    cc1,cc2

CONCRETE_VARIABLES
    cv1

// Constants typed in properties
PROPERTIES
    cc1 : rust_u32 &
    cc2 : rust_bool

// Variables typed in invariant
INVARIANT
    cv1 : rust_i16

INITIALISATION
    cv1 := 1

END
```

```
IMPLEMENTATION atomic_type_i
REFINES atomic_type

SEES
    b2rust_types

// Another variable or constant can be declared in refinement
CONCRETE_VARIABLES
    cv2

INVARIANT
    //cv1 was typed in abstract machine, however it can be retyped
    cv1 : rust_i8 &

    cv2 : rust_i16

VALUES
    cc1 = 11 ;
    cc2 = FALSE

INITIALISATION
    cv1 := 1 ;
    cv2 := 2

END
```

FIGURE 2 – Atomic type

Translation to Rust :

```rust
use std::convert::TryFrom;

pub struct atomic_type {
    // Concrete variables & constants.
    pub r#cv1: i8,
    pub r#cv2: i16,
}

impl Default for atomic_type {
    fn default() -> Self {
        let mut instance = Self {
            r#cv1: i8::default(),
            r#cv2: i16::default(),
        };
        instance.initialisation();
        instance
    }
}
impl atomic_type {
    // Constant's `VALUES`.
    pub const r#cc1: u32 = 11;
    pub const r#cc2: bool = false;
    fn initialisation(&mut self) {
        // `INITIALISATION` clause.
        self.r#cv1 = i8::try_from(1).unwrap();
        self.r#cv2 = i16::try_from(2).unwrap();
    }
}
```

FIGURE 3 – Translated atomic type

Tabular type

Every concrete data ‘ident’ intended to be arrays must be typed as :

ident : [interval “*”]+ interval –> rust_type

while :

Non terminal	Productions
interval	: := 0..ExpressionArith | rust_integer
rust_array	: := [interval "*"]+ interval "–>" rust_type

See the definition of Expression Arith

Note : b2rust is not able to interpret the **supremum** of the interval. Therefore, if the interval is empty (ExpressionArith <0), b2rust will not generate an empty array, but negative size array, which makes no sense, and the error will be told at compilation. In addition, the upper bound of the interval is not allowed to be a concrete_variable. (However, it is possible to create an empty array with the interval 0..-1)

B code example :

```
MACHINE
    main_array_15

SEES
    b2rust_types

CONCRETE_CONSTANTS
    tab, tab_size

PROPERTIES
    tab_size  : rust_u8 &
    tab : (0..1) --> ((0..tab_size) * rust_u8  -->  rust_i8)


END
```

```
IMPLEMENTATION main_array_15_i
REFINES main_array_15

SEES
    b2rust_types

VALUES
    tab_size = 8 ;
    tab = {0|-> (0..tab_size) * rust_u8 * {0},
1|-> (0..tab_size) * rust_u8 * {0}}


 END
```

FIGURE 4 – Array type

Translation to Rust :

```rust
use std::convert::TryFrom;

pub struct main_array_15 {}

impl Default for main_array_15 {
    fn default() -> Self {
        let mut instance = Self {};
        instance.initialisation();
        instance
    }
}
impl main_array_15 {
    // Constant's `VALUES`.
    pub const r#tab_size: u8 = 8;
    pub const r#tab: [[[i8; (255 + 1) as usize]; (Self::r#tab_size + 1) as usize];
        (1 + 1) as usize] = [
        [[0; (255 + 1) as usize]; (Self::r#tab_size + 1) as usize],
        [[0; (255 + 1) as usize]; (Self::r#tab_size + 1) as usize],
    ];
    fn initialisation(&mut self) {}
}
```

FIGURE 5 – Translated array type

Set

Sets introduce **new types** recognized by b2rust. Sets will be translated into rust **enumerations**. For each set that has 'set_ident' as identifier and each concrete data that has 'ident' as identifier, the typing

> ident : set_ident

is accepted in b2rust.

B code example :

```
MACHINE
    set_type

SEES
    b2rust_types

SETS
    CAT = {MaineCoon, Siamese, Tiger}

CONCRETE_CONSTANTS
    cc, tabCat

PROPERTIES
    cc : CAT &
    tabCat : 0..5 --> CAT

END
```

```
IMPLEMENTATION set_type_i
REFINES set_type

SEES
    b2rust_types

VALUES
    cc = MaineCoon ;
    tabCat = (0..5)*{Tiger}


 END
```

FIGURE 6 – Set type

Translation to rust :

```rust
use std::convert::TryFrom;

#[derive(Clone, Copy, Default, Debug, Eq, PartialEq)]
pub enum CAT {
    #[default]
    MaineCoon,
    Siamese,
    Tiger,
}

pub struct set_type {}

impl Default for set_type {
    fn default() -> Self {
        let mut instance = Self {};
        instance.initialisation();
        instance
    }
}
impl set_type {
    // Constant's `VALUES`.
    pub const r#cc: CAT = CAT::MaineCoon;
    pub const r#tabCat: [CAT; (5 + 1) as usize] = [CAT::Tiger; (5 + 1) as usize];
    fn initialisation(&mut self) {}
}
```

FIGURE 7 – Translated set type

Conclusion et Extension

Non terminal	Productions
rust_type	: := rust_integer | rust_bool | rust_string | rust_array | set_ident

In the case where a type not defined in b2rust_types wants to be translated as a type of b2rust_types, it is possible to extend the syntax by adding associations in the file ‘b2rust_types.cfg’ file. For example : the user has defined uint8_t of C in B machine. He would like b2rust to translate

```
vv : uint8_t
```

into

```
vv : u8
```

He just has to add the pair ’ uint8_t rust_u8’ in b2rust_types.cfg.

See an example

Operations

Parameters

All **input** and **output** operation parameters must be typed once and only once in the **precondition** of the **abstraction**. If the operation has no parameters, there is no need to start the operation with a 'PRE'.

B code example :

```
MACHINE
    op_type

SEES b2rust_types

CONCRETE_VARIABLES
    cv

INVARIANT
    cv : rust_i8

INITIALISATION
    cv := 10

OPERATIONS

    res <-- op(tab,index,cc) =
    PRE
        // Type operation parameters
        res : rust_i32 &
        index : rust_u8 &
        tab : rust_u8 --> rust_i32 &
        cc : rust_i32 &

        //You can still have other conditions among the preconditions
        tab(0) > cc
    THEN
        //whatever postcondition you want, even 'skip'

        // You don't have to type cv even if you use it in your
        // operation body, because cv is not a parameter
        res := tab(index) + cc + cv
    END;

    decr =
    BEGIN
        skip
    END

END
```

```
IMPLEMENTATION op_type_i
REFINES op_type

SEES
    b2rust_types

INITIALISATION
    cv := 10

OPERATIONS

    res <-- op(tab,index,cc) =
    BEGIN
        res := tab(index) + cc
    END
    ;
    decr =
    BEGIN
        cv := cv -1
    END
END
```

Figure 8 – Op type

Translation to Rust :

Local variables

Local variables in implementation must be typed with the **vv : :E** BecomesIn or **vv :(vv :E)** BecomesSuchThat operator in the first instructions after being declared.

```rust
use std::convert::TryFrom;

pub struct op_type {
    // Concrete variables & constants.
    pub r#cv: i8,
}

impl Default for op_type {
    fn default() -> Self {
        let mut instance = Self {
            r#cv: i8::default(),
        };
        instance.initialisation();
        instance
    }
}
impl op_type {
    fn initialisation(&mut self) {
        // `INITIALISATION` clause.
        self.r#cv = i8::try_from(10).unwrap();
    }

    pub fn op(
        &mut self,
        r#tab: &[i32; (255 + 1) as usize],
        r#index: &u8,
        r#cc: &i32,
        r#res: &mut i32,
    ) {
        *r#res = i32::try_from(((r#tab[(*r#index) as usize]) + (*r#cc))).unwrap();
    }

    pub fn decr(&mut self) {
        self.r#cv = i8::try_from(((self.r#cv) - (1))).unwrap();
    }
}
```

FIGURE 9 – translated Op type

B code example :

```
MACHINE
    local_type
SEES
    b2rust_types

SETS
   SURTYPE = {toto,tata,titi}

OPERATIONS
   op(tab1, tab2) =
   PRE
      tab1 : (0..10) * (0..1) --> SURTYPE &
      tab2 : (0..1) --> SURTYPE
   THEN
      skip
   END
END
```

```
IMPLEMENTATION local_type_i

REFINES
   local_type

SEES b2rust_types

OPERATIONS
   op(tab1,tab2) =
   VAR
      loc1,loc2, xx, yy
   IN
      // If the BecomesIn and BecomesSuchThat instruction showed error, it
      // means you did not modify the ressource file AtelierB of your project.
      // Please check the first section of the usermanual
      xx :: rust_u8;
      yy : (yy : rust_u8);
      loc1 :(loc1: SURTYPE);
      loc2 :: SURTYPE;
      xx := 1;
      yy := 1;
      loc1:= tab1(xx,yy);
      loc2:=tab2(yy)
   END
END
```

FIGURE 10 – Op type

Translation to Rust :

Expressions

Non terminal	Productions
Expression	: := ExpressionArith
	| ExpressionTabular

Non terminal	Productions
	| ExpressionBoolean | TermeSimple

Non terminal	Productions
ExpressionBoolean	: := BooleanLiteral | “bool”“(” Condition “)”

Non terminal	Productions
TermeSimple	: := Iden_ren | IntegerLitteral | BooleanLiteral | “bool”“(” Condition “)” | SetElement

SetElement refers to the declared sets’s elements.

```rust
use std::convert::TryFrom;

#[derive(Clone, Copy, Default, Debug, Eq, PartialEq)]
pub enum SURTYPE {
    #[default]
    toto,
    tata,
    titi,
}

pub struct local_type {}

impl Default for local_type {
    fn default() -> Self {
        let mut instance = Self {};
        instance.initialisation();
        instance
    }
}
impl local_type {
    fn initialisation(&mut self) {}

    pub fn op(
        &mut self,
        r#tab1: &[[SURTYPE; (1 + 1) as usize]; (10 + 1) as usize],
        r#tab2: &[SURTYPE; (1 + 1) as usize],
    ) {
        {
            let mut r#xx: u8 = Default::default();
            let mut r#yy: u8 = Default::default();
            let mut r#loc1: SURTYPE = Default::default();
            let mut r#loc2: SURTYPE = Default::default();
            r#xx = u8::try_from(1).unwrap();
            r#yy = u8::try_from(1).unwrap();
            r#loc1 = SURTYPE::try_from(r#tab1[r#xx as usize][r#yy as usize]).unwrap();
            r#loc2 = SURTYPE::try_from(r#tab2[(r#yy) as usize]).unwrap();
        }
    }
}
```

FIGURE 11 – translated Op type

Arithmetic expressions

Lambda functions Rust has two constraints for arithmetic expressions :

- **Compilation** : Operands must have the same rust type, with the exception of :
  - **left-shift** : the second operand must be of **rust_unsigned_integer** type.
  - **right-shift** : the second operand must be of **rust_unsigned_integer** type
  - **exponentiation** : the second operand must be of **u32** type.
- **Execution** : Rust panics at execution when there is an **overflow** (although there are options to disable this).

b2rust respect rust's choice. To check that there is no overflow in AtelierB, lambdas functions have been provided in 'b2rust_types.mch' to modeling arithmetic operators.

Non terminal	Productions
ExpressionArith	: := ExpressionArith "+" ExpressionArith | ExpressionArith "-" ExpressionArith | ExpressionArith "*" ExpressionArith

Non terminal	Productions
	| ExpressionArith “/” ExpressionArith
	| ExpressionArith “mod” ExpressionArith
	| ExpressionArith “**” ExpressionArith
	| - (ExpressionArith)
	| add “_” dom “(” ExpressionArith “,” ExpressionArith “)”
	| sub “_” dom “(” ExpressionArith “,” ExpressionArith “)”
	| mul “_” dom “(” ExpressionArith “,” ExpressionArith “)”
	| div “_” dom “(” ExpressionArith “,” ExpressionArith “)”
	| mod “_” dom “(” ExpressionArith “,” ExpressionArith “)”
	| pow “_” dom “(” ExpressionArith “,” ExpressionArith “)”
	| lshift “_” dom “(” ExpressionArith “,” ExpressionArith “)”
	| rshift “_” dom “(” ExpressionArith “,” ExpressionArith “)”
	| and “_” dom “(” ExpressionArith “,” ExpressionArith “)”
	| or “_” dom “(” ExpressionArith “,” ExpressionArith “)”
	| xor “_” dom “(” ExpressionArith “,” ExpressionArith “)”
	| “(” ExpressionArith “)”
	| ident
	| integer_literal
dom	: := “i8”
	| “i16”
	| “i32”
	| “i64”
	| “i128”
	| “u8”
	| “i16”
	| “i32”
	| “i64”
	| “i128”

These lambdas functions have the particular feature of :

1. generate **proof obligations** on the operand type as well as on the result

type
2. perform **conversion** of operands to result type (justified if code has been proven).

The advantage of conversion is that it would be possible to perform an operation with two operands of different (but compatible) type.

To summarize,

Operand s	B Code	Translated Rust Code	Result
aa :i8 = 120 bb :i8 = 7	aa + bb	aa + bb	OK
aa :i8 = 120 bb :u8 = 7	aa + bb	aa + bb	compile error (not same type)
aa :i8 = 120 bb :u8 = 7	add_i8(aa,bb)	i8 : :try_into(aa).unwrap() + i8 : :try_into(bb).unwrap()	OK
aa :i16 = 128 bb :u8 = 7	pow_i8(aa,bb)	(i8 : :try_into(aa).unwrap()) .pow(bb as u32)	panic (conversion failed)
aa :i16 = 128 bb :i8 = 2	lshift_u32(aa,bb)	u32 : :try_into(aa).unwrap() » b as usize	OK

Conversion b2rust uses two types of conversion :

- 'as type' is an explicit conversion, i.e. a bit-by-bit reinterpretation **without verification**.
- 'type : :try_into(ident).unwrap()' is a conversion **with verification**, rust panics if the conversion fails.

b2rust always adds conversions with verification when lambda functions are used, except in the following cases :

- VALUES clause
- Second argument of exponentiation
- Second argument of left shift
- Second argument of right shift

The explicit conversion 'as' will be used. The danger of using 'as' is that rust doesn't panic if the conversion fails.

For example :

```
let aa : i16 = 128
let bb : i8 = aa as i8 // bb is -128
```

Therefore it is highly recommended to validate proof obligations before translating to prevent this kind of situation.

Extension In the case where the user wants to that a lambda function to be translated to a Rust operator, it is possible to extend the syntax by adding associations in the file 'b2rust_operations.cfg' file. For instance, to direct b2rust to translate `bitwise_and_uint32` defined in a B machine to Rust's `aa ^ bb`, add the pair ' bitwise_and_uint32 and_u32' in configuration file b2rust_operations.cfg.

See an example

Tabular Expression

Non terminal	Productions
ExpressionTabular	: := “{” (IntegerLitteral “|->” Expression)+, “}” | (interval)+* interval * “{” ExpressionArith “}” | (interval)+* interval * “{” ExpressionBoolean “}” | (interval)+* interval * “{” TermeSimple “}” | ident

Note that IntegerLitteral must be positive.

Instructions

B0 instructions are translatable.

Assignments

When an assignment of the form occurs :

ident := Expression

b2rust automatically adds conversions with verification 'try_into(_).unwrap()' to convert the type of the expression to the type of ident.

This conversion is useful when the expression has a type that is **compatible** but not identical to ident.

B code example :

```
MACHINE
    main_weird_01

SEES
    b2rust_types


OPERATIONS
    res <-- op(aa) =
    PRE
        res : rust_i8 &
        aa : rust_i16
    THEN
        res := aa
    END
END
```

```
IMPLEMENTATION main_weird_01_i
REFINES main_weird_01

SEES
    b2rust_types

OPERATIONS

     res <-- op(aa)=
     BEGIN
         res := aa
     END
 END
```

FIGURE 12 – Conversion Example

Translation to Rust :

```rust
use std::convert::TryFrom;

pub struct main_weird_01 {}

impl Default for main_weird_01 {
    fn default() -> Self {
        let mut instance = Self {};
        instance.initialisation();
        instance
    }
}
impl main_weird_01 {
    fn initialisation(&mut self) {}

    pub fn op(&mut self, r#aa: &i16, r#res: &mut i8) {
        *r#res = i8::try_from(*r#aa).unwrap();
    }
}
```

FIGURE 13 – Translated Conversion Example

Exception : It is not possible to convert **array** types (but array element is accepted).

If the conversion fails, rust panics at runtime. To avoid this problem, lambda functions fit (identity on a domain) are provided. For each assignment, it is recommended to use fit to ensure that the assigner has a type compatible with the assignable.

b2rust ignore fit lambdas function, for example :

> ident := fit_i8(Expression)

is translated into

> ident = Expression

The syntax of the fit can also be extended, as can the operations. Note that the suffix of fit lambda function is useless, you can push your self made fit lambda function with any fit functions defined in b2rust_types.mch.

See how to extend syntax operations

Operation call

No constraints in particular, just an explanation of how the function call is translated. The idea is simple :

1. copy the input and output parameters
2. make a function call on these copies
3. modify the output parameters with the modified copy.

This is a mechanism for avoiding the borrowing problem in Rust.

Translation to Rust :

```
MACHINE
    localop_type

SEES
    b2rust_types

CONCRETE_VARIABLES
    cv1,cv2

INVARIANT
    cv1 : rust_i8 &
    cv2 : rust_i8

INITIALISATION
    cv1 := 1 ||
    cv2 := 2

OPERATIONS
    swap =
    BEGIN
        cv1:= cv2||
        cv2:= cv1
    END

END
```

```
IMPLEMENTATION localop_type_i
REFINES localop_type

SEES
    b2rust_types


INITIALISATION
    cv1 := 1;
    cv2 := 2

LOCAL_OPERATIONS

    res1,res2 <-- identity(param1,param2) =
    PRE
        res1 : rust_i8 & param1 : rust_i8 &
        res2: rust_i8 & param2 : rust_i8
    THEN
        res1 := param1 ||
        res2 := param2
     END

OPERATIONS
    res1,res2 <-- identity(param1,param2) =
    BEGIN
        res1 := param1;
        res2 := param2
    END;

    swap =
    BEGIN
        cv2,cv1 <-- identity(cv1,cv2)
     END
 END
```

FIGURE 14 – Example of Operation Call

Referenced machine

Non terminal	Productions
Clause_imports	: := "IMPORTS" ((Ident_ren "["(" Instanciation +",")]")+)
Clause_sees	: := "SEES" Ident_ren+
Clause_extends	: := EXTENDS" (Ident ["(" Instanciation +",")]+",")+"," ")"])

Every referenced machine (imported, seen, extended) becomes a field of a struct.

B code example :

Translation to Rust :

```rust
use std::convert::TryFrom;

pub struct localop_type {
    // Concrete variables & constants.
    pub r#cv1: i8,
    pub r#cv2: i8,
}

impl Default for localop_type {
    fn default() -> Self {
        let mut instance = Self {
            r#cv1: i8::default(),
            r#cv2: i8::default(),
        };
        instance.initialisation();
        instance
    }
}
impl localop_type {
    fn initialisation(&mut self) {
        // `INITIALISATION` clause.
        self.r#cv1 = i8::try_from(1).unwrap();
        self.r#cv2 = i8::try_from(2).unwrap();
    }

    fn identity(&mut self, r#param1: &i8, r#param2: &i8, r#res1: &mut i8, r#res2: &mut i8) {
        *r#res1 = i8::try_from(*r#param1).unwrap();
        *r#res2 = i8::try_from(*r#param2).unwrap();
    }

    pub fn swap(&mut self) {
        {
            let mut r#inputCopy1 = self.r#cv1 as i8;
            let mut r#inputCopy2 = self.r#cv2 as i8;
            let mut r#outputCopy1 = self.r#cv2 as i8;
            let mut r#outputCopy2 = self.r#cv1 as i8;
            self.identity(
                &r#inputCopy1,
                &r#inputCopy2,
                &mut r#outputCopy1,
                &mut r#outputCopy2,
            );
            self.r#cv2 = i8::try_from(r#outputCopy1).unwrap();
            self.r#cv1 = i8::try_from(r#outputCopy2).unwrap();
        }
    }
}
```

FIGURE 15 – Translated operation call

```
MACHINE
    import_type

INCLUDES
    imported1

OPERATIONS
    op = skip

END
```

```
IMPLEMENTATION import_type_i
REFINES import_type

SEES
    seen

IMPORTS
    imported1

EXTENDS
    imported2


END
```

FIGURE 16 – Referenced Machine

```
mod imported1;
mod imported2;
mod seen;

use std::convert::TryFrom;

pub struct import_type {
    // Instances of imported modules.
    pub _1_imported1: imported1::imported1,
    pub _2_seen: seen::seen,
    pub _3_imported2: imported2::imported2,
}

impl Default for import_type {
    fn default() -> Self {
        let mut instance = Self {
            // Instances of imported modules initialization.
            _1_imported1: Default::default(),
            _2_seen: Default::default(),
            _3_imported2: Default::default(),
        };
        instance.initialisation();
        instance
    }
}
impl import_type {
    fn initialisation(&mut self) {
        // Instances of imported modules.
    }

    pub fn op(&mut self) {
        self._3_imported2.op();
    }
}
```

Figure 17 – Translated referenced machine

Formal parameters

Non terminal	Productions
Instanciation	: := TermeSimple | ExpressionArith | ExpressionBoolean

The machine's formal parameters must be **typed** in the **INVARIANT** clause. Renaming is accepted as long as there is only one renaming prefix.

In this version of AtelierB, there are still bugs with multiple renaming prefixes in atelierB.

The translation choice for the formal parameters is to add a **private field** in the Rust struct. Then add a constructor named **new** in addition to the default constructor. Machines with parameters will be instantiated using new.

B code example :

```
MACHINE
    imported(param)

SEES
    b2rust_types

CONSTRAINTS
    param : NAT

CONCRETE_CONSTANTS
    cc

PROPERTIES
    cc : rust_i8

CONCRETE_VARIABLES
    cv

INVARIANT
    cv : rust_i8

INITIALISATION
    cv := param

END
```

```
IMPLEMENTATION imported_i(param)

REFINES imported

SEES
    b2rust_types

INVARIANT
    param : rust_i8 //type your formal parameter here

VALUES
    cc = 10

INITIALISATION
    cv := param

 END
```

FIGURE 18 – Formal parameters

Translation to Rust :

```rust
use std::convert::TryFrom;

pub struct imported {
    // Parameters.
    r#param: i8,
    // Concrete variables & constants.
    pub r#cv: i8,
}

impl Default for imported {
    fn default() -> Self {
        let mut instance = Self {
            r#param: i8::default(),
            r#cv: i8::default(),
        };
        instance.initialisation();
        instance
    }
}
impl imported {
    // Constant's `VALUES`.
    pub const r#cc: i8 = 10;
    pub fn new(r#param_arg: i8) -> Self {
        let mut instance = Self {
            r#param: r#param_arg,
            r#cv: i8::default(),
        };
        instance.initialisation();
        instance
    }
    fn initialisation(&mut self) {
        // `INITIALISATION` clause.
        self.r#cv = i8::try_from(self.r#param).unwrap();
    }
}
```

```rust
mod imported;

use std::convert::TryFrom;

pub struct param_type {
    // Instances of imported modules.
    pub _1_M1: imported::imported,
    pub _2_M2: imported::imported,
    pub _3_imported: imported::imported,
}

impl Default for param_type {
    fn default() -> Self {
        let mut instance = Self {
            // Instances of imported modules initialization.
            _1_M1: Default::default(),
            _2_M2: Default::default(),
            _3_imported: Default::default(),
        };
        instance.initialisation();
        instance
    }
}
impl param_type {
    fn initialisation(&mut self) {
        // Instances of imported modules.
        self._1_M1 = imported::imported::new(10);
        self._2_M2 = imported::imported::new(15);
        self._3_imported = imported::imported::new(5);
    }
}
```

FIGURE 19 – Translated formal parameters

Basic module

For machines without an implementation, b2rust generates a file with the extension '.rs.template', which serves as a **template**.

In the template content :

— concrete_variables sometimes translated
— concrete_constants sometimes translated and always commented.
— Translatable instructions in initialization are sometimes translated.
— Operation signatures are **translated**, but the operation body only has a **unimplemented !** macro.

```
MACHINE
    error

OPERATIONS
    error_msg(message) =
    PRE
        message : STRING
    THEN
        skip
    END
END
```

```rust
use std::convert::TryFrom;

pub struct error {
}

impl Default for error {
    fn default() -> Self {
        let mut instance = Self {
};
instance.initialisation();
instance}
}
impl error {
fn initialisation(&mut self) {
}

pub fn error_msg(&mut self, r#message: &str) {
unimplemented!("error_msg is unimplemented");
}
}
```

FIGURE 20 – Base

A bash file **check.sh** is provided to verify that the user has implemented the struct and associated methods.

File not intended for translation

There are B machines whose only purpose is to serve as a **library** to provide typing information and lambdas functions.These machines are not intended to be translated, but to generate proof obligations, such as 'b2rust_types.mch'. To manage this kind of file, b2rust provides a configuration file 'b2rust_exceptions.cfg', the machines inside this file will not be **seen** in the translation by the other machines (they don't become struct fields).

However, every referenced type in the library that needs to be translated must have an association in b2rust_types.cfg. Each lambda function used must have an association in b2rust_operations.cfg

Example in B :

```
MACHINE
    c4b_types


CONCRETE_CONSTANTS
    bitwise_sll_uint8,
    add_uint32,
    sub_uint32,
    fit_in_u8,

    uint8_t,
    uint16_t,
    uint32_t,
    MAX_UINT32, //not translated, because not typed
    MAX_UINT16, //not translated, because not typed
    MAX_UINT8   //not translated, because not typed

PROPERTIES
    MAX_UINT32 = 4294967295&
    MAX_UINT16 = 65535 &
    MAX_UINT8 = 255 &
    uint32_t = 0..4294967295 &    //same def with rust_u32, put "uint32_t rust_u32" in
                                  // b2rust_types.cfg, then aa : uint32_t <=> aa : rust_u32
    uint16_t = 0..65535 &
    uint8_t = 0..255 &

    bitwise_sll_uint8 : uint8_t*uint8_t --> uint8_t &
    add_uint32 : uint32_t*uint32_t --> uint32_t &
    sub_uint32 : uint32_t*uint32_t --> uint32_t &
    fit_in_u8 : uint8_t --> uint8_t &

    bitwise_sll_uint8 = %(x1,x2).(x1 : uint8_t & x2 : uint8_t | (x1 * (2**x2)) mod (MAX_UINT8 +
1)) &

    add_uint32 = %(x1,x2).(x1 : uint32_t & x2 : uint32_t | (x1 + x2) mod (MAX_UINT32 + 1)) &
    sub_uint32 = %(x1,x2).(x1 : uint32_t & x2 : uint32_t | (x1 - x2 + MAX_UINT32 + 1) mod
(MAX_UINT32 + 1)) &

    fit_in_u8 = %(xx).(xx : uint8_t | xx)
END
```

FIGURE 21 – Import

Translation to Rust :

A warning will be triggered if b2rust doesn't recognize the typing information of a concrete data.

```
MACHINE
    oprust_types

SEES
    b2rust_types,c4b_types

OPERATIONS
    res <-- lshift(aa,bb) =
    PRE
        // uint8_t type is not recognised by b2rust,
        // unless it has an association with a type of
        // b2rust_types, check b2rust_types.cfg
        aa : uint8_t &

        bb : rust_u8 &
        res : rust_u8
    THEN
        res :: uint8_t
    END

END
```

```
IMPLEMENTATION oprust_types_i
REFINES oprust_types


SEES
    b2rust_types, c4b_types


OPERATIONS
    res <-- lshift(aa,bb) =
    BEGIN
        // bitwise_sll_uint8 is a lambda function
        // not recognized by b2rust,
        // unless it has an association with
        // a lambda funciton defined in  b2rust_types, check
        // b2rust_operations.cfg

        // same for fit_in_u8
        res := fit_in_u8(bitwise_sll_uint8(aa,bb))
    END


 END
```

FIGURE 22 – Import

```
use std::convert::TryFrom;

pub struct oprust_types {}

// no field c4b and b2rust_types, because there are in b2rust_exceptions.cfg
impl Default for oprust_types {
    fn default() -> Self {
        let mut instance = Self {};
        instance.initialisation();
        instance
    }
}
impl oprust_types {
    fn initialisation(&mut self) {}

    // fit disappeared, it is good
    // bitwise_sll_uint8_t is considered as lshift_u8
    pub fn lshift(&mut self, r#aa: &u8, r#bb: &u8, r#res: &mut u8) {
        *r#res = u8::try_from(((u8::try_from(*r#aa).unwrap()) << (*r#bb as usize))).unwrap();
    }
}
```

FIGURE 23 – translated Import

Conclusion

For b2rust to generate code, you need to ask yourself the following questions :

- Are all concrete_constants and concrete_variables typed in the right clause ?
- Do all operations have their input and output parameters typed in the abstract machine precondition ?
- Are local variables typed with BecomesIn or BecomesSuchThat ?
- Are the associations between my types and my function lambdas with those of b2rust_types done correctly ?