CLEARSY

Safety Solutions Designer

b2rust

User Manual

Christophe CHEN

2023-07-17

Contents

1 Introduction

Ce document sert de **manuel utilisateur** pour **b2rust**. Il présente ses **contraintes d'utilisation** et les **choix des traductions** des modèles B en Rust.

2 Lancer b2rust

2.1 Configuration

2.1.1 Fichier ressource

Ajouter les lignes suivantes dans le fichier ressource **AtelierB** de votre projet :

```
1    ATB*BCOMP*Allow_Becomes_Member_Of:TRUE
2    ATB*BCOMP*Allow_Becomes_Such_That:TRUE
3    ATB*TC*Allow_Becomes_Member_Of:TRUE
4    ATB*TC*Allow_Becomes_Such_That:TRUE
```

Ces lignes sont nécessaires pour générer le `bxml` des implementations qui utilisent les instructions :
- **vv :: E** (Becomes in)
- **vv : (vv : E)** (Becomes such that)

2.1.2 Fichier de configuration

b2rust a besoin de connaître le répertoire où se trouvent ses trois fichiers de configuration :
-**b2rust_types.cfg**
-**b2rust_operations.cfg**
-**b2rust_exceptions.cfg**.

Par défaut, ces trois fichiers de configuration sont présents dans le répertoire **files** de b2rust. Lancez la commande :

```
1  export B2RUST_CONF_HOME=~/path/to/b2rust/files
```

Si vous avez besoin de modifier la configuration de b2rust, vous pouvez copier ces fichiers et les rééditer dans un autre répertoire. N'oubliez pas de mettre à jour le chemin.

2.2 Génération du code

2.2.1 Génération du bxml

b2rust traduit en fait le `bxml`. Un fichier script `gen_bxml.sh` est fourni pour créer le `bxml` à partir des fichiers d'extension **mch**, **ref** et **imp**.

Pour exécuter ce script, il est recommandé d'ajouter le chemin vers l'exécutable **bxml** d'AtelierB :

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

Le script **gen_bxml** peut maintenant être exécuté :

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

— **$1** : le chemin vers le répertoire contenant l'exécutable **bxml** d'AtelierB

— **$2** : le répertoire cible contenant les fichiers dont on veut générer le `bxml`

— **$3** : le fichier ressource d'AtelierB

2.2.2 Génération du code Rust

Après la génération du `bxml`, **b2rust** serait capable de traduire le B.

Pour traduire une machine B, lancez la commande :

```
1 ./b2rust  $1 -I $2 [-O $3]
```

— **$1** : le nom de la machine que vous voulez traduire.
— **$2** : le répertoire cible
— **$3** : le répertoire où **b2rust** génère ces fichiers (facultatif)

Cette commande traduit aussi récursivement toutes les machines vues, importées et étendues par la machine implementation.

Illustration :

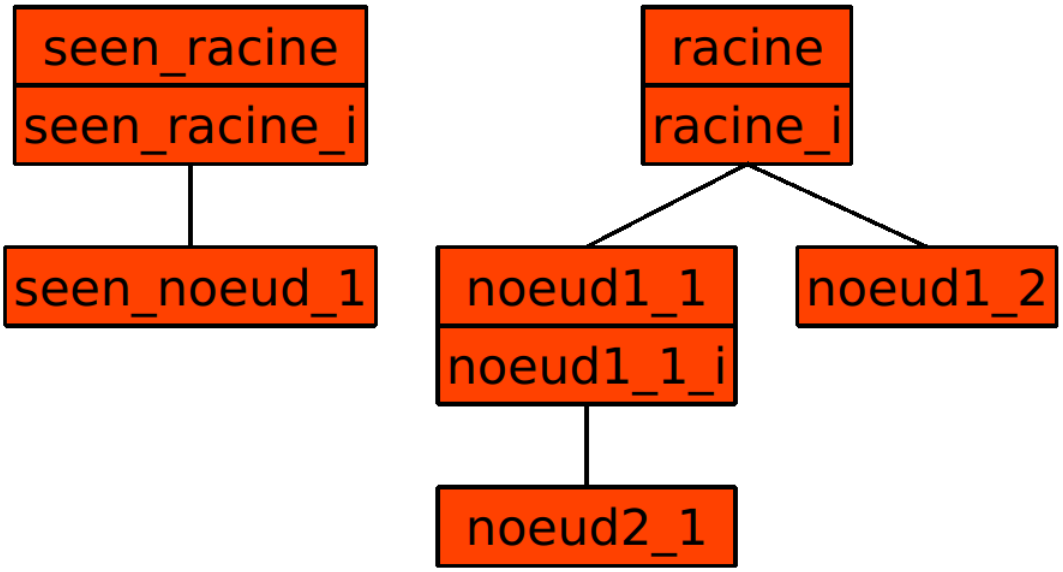

Figure 1 – Translation tree

Par exemple :
- b2rust appliqué à la machine noeud1_1 traduit les machines de son sous-arbre noeud1_1 et noeud2_1 mais pas les machines racine, noeud1_2.
- Dans le cas où seen_racine est vu par une des machines de sous-arbre de noeud1_1, les machines seen_racine et seen_noeud sont également traduit.

3 Code B traduisible par b2rust

b2rust ne traduit que l'**implementation** des machines et les **modules de bases**.
Chaque machine est traduit en struct rust.
Les concrete_variables et les machines référencés deviennent les champs du struct.
Les concrete_constants deviennent les constantes statiques du struct.
Les operations de la machine deviennent des methodes du struct.

3.1 Le typage

b2rust ne traduit que les concrete_constants et concrete_variables qu'on appelera données concretes.
Chaque donnée doit être typé au moins une fois dans l'ensemble du module pour être traduit:
- Les **constantes** doivent etre typé dans la clause **PROPERTIES**
- Les **variables** doivent etre typé dans la clause **INVARIANT**

Toutes les données concrètes d'identifiant 'ident' doivent etre typé en utilisant l'opérateur appartient '**:**' belongsTo de la forme :

```
1  ident : rust_type
```

b2rust determine le type de la donné en fonction de l'identifiant de rust_type, pour simplifier il est preferable que toute les machines voient b2rust_types.mch qui fournissent déjà une bonne définition de chaque identifiant.

Chaque donnée ne peut pas être typé plusieurs fois au sein d'une même machine, mais ils peuvent être typé différemment entre machines abstraite et ses raffinement. Seul le dernier typage est retenu pour b2rust.

3.1.1 Type atomique

Non terminal	Productions
rust_integer	::= rust_signed_integer | rust_unsigned_integer
rust_bool	::= TRUE | FALSE

Non terminal	Productions
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

Il faut noter que atelierB accepte le typage de string qu’en paremetre d’entrée d’opération.

Exemple d’un code B :

```
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

Et sa traduction en 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

3.1.2 Type tableau

Les données concretes d'identifiant 'ident' destinés à être des tableaux doivent être typé de forme :

```
1 ident : [interval "*"]+ interval --> rust_type
```

où :

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

Non terminal	Productions
ExpressionArith	::= ExpressionArith "+" ExpressionArith | ExpressionArith "-" ExpressionArith | ExpressionArith "*" ExpressionArith | ExpressionArith "/" ExpressionArith | "(" ExpressionArith ")" | ident | integer_literal

Remarque : b2rust n'est pas capable d'evaluer la valeur de la **borne sup** de l'interval. Par conséquent si l'interval est vide (ExpressionArith < 0) , b2rust ne va pas générer un tableau vide mais un tableau de taille negatif ce qui n'a aucun sens, l'erreur sera signalé à la compilation du code rust. De plus, la borne sup de l'intervalle n'a pas le droit d'être une concrete_variable. (Cependant il est possible de créer un tableau vide avec l'intervalle 0..-1.)

Exemple en B :

```
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

Et sa traduction en 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

3.1.3 Set

Les set introduisent des **nouveaux types** reconnu par b2rust. Les sets seront traduit en **enumerations** rust.
Pour chaque set d'identifiant set_ident et chaque donnée concrete d'identifiant ident, le typage :

```
ident : set_ident
```

est accepté dans b2rust

Exemple en B :

```
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

Et sa traduction en 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

3.1.4 Conclusion et Extension

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

Dans le cas où un type non-défini dans b2rust_types veut etre traduit comme étant un type de b2rust_types, il est possible d'etendre la synthaxe de typage en ajoutant des associations dans le fichier 'b2rust_types.cfg'. Par exemple : l'utilisateur a défini uint8_t de C dans une machine B. Il souhaite que b2rust traduit

```
1    vv : uint8_t
```

en

```
1    vv : u8
```

Il suffit qu'il ajoute la paire ' uint8_t rust_u8' dans b2rust_types.cfg.

Voir un exemple

3.2 Les operations

3.2.1 Parametres

Tout les paramètres d'**entrées** et **sorties** des opérations doivent être typé une et une seule fois dans la **precondition** de l'**abstraction**.
Si l'operation n'a aucun paramètres, il n'est pas necessaire de commencer l'operation par un 'PRE'.

Exemple en B :

```
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

Et sa traduction en rust :

```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

3.2.2 Variables locales

Les variables locales dans l'implementation doivent etre typé avec l'operateur **vv::E** BecomesIn ou **vv:(vv:E)** BecomesSuchThat dans les premieres instructions apres avoir été declarés.

Exemple en B :

```
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

Et sa traduction en rust :

```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

3.3 Expressions

Non terminal	Productions
Expression	::= ExpressionArith
	| ExpressionTableau
	| ExpressionBoolenne
	| TermeSimple

SetElement désigne les elements des sets déclarés.

Non terminal	Productions
Expression_booléenne	::= Booléen_lit
	| “bool”“(” Condition “)”

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

3.3.1 Expressions arithmetiques

Lambda fonctions Rust impose deux contraintes pour les expressions arithmétiques :
- **Compilation** : Les opérandes doivent avoir le même type rust, à l'exception du :
- **left-shift** : la seconde opérande doit être de type **rust_unsigned_integer**
- **right-shift** : la seconde opérande doit être de type **rust_unsigned_integer**
- **exponentiation** : la seconde opérande doit être de type **u32**
- **Exécution** : Rust panique à l'exécution lorsqu'il y'a un **overflow** (bien qu'il y'a des options pour désactiver ceci).

b2rust a choisi de respecter le choix de rust.

Pour controler qu'il n'y'a pas d'overflow dans atelierB, des lambdas fonctions ont été fourni dans 'b2rust_types.mch' afin de modeliser des opérateurs arithmetiques.

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

Non terminal	Productions
	| “i64”
	| “i128”
	| “u8”
	| “i16”
	| “i32”
	| “i64”
	| “i128”

Ces lambdas fonctions ont la particularité de :
1) générer des **obligations de preuve** sur le type des opérandes ainsi sur le type du resultat
2) faire la **conversion** des opérandes vers le type resultat (justifié si code prouvé).

L'intérêt de faire la conversion c'est qu'il serait possible de faire une opération avec deux opérandes de type différent (mais compatible).

En résumé,

Operandes	Code B	Code Rust traduit	Résultat
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 utilise deux types de conversions:
- 'as type' s'agit d'une conversion explicite, c'est à dire une réinterpretation bit à bit **sans verification**.
- 'type::try_into(ident).unwrap()' s'agit d'une conversion **avec verification**, rust panique si la conversion echoue.

b2rust ajoute systématiquement les conversions avec vérification lorsque les lambdas fonctions sont utilisé à l'exception des cas suivants:
- Clause VALUES
- Deuxieme argument de l'exponentiation
- Deuxieme argument du left shift
- Deuxieme argument du right shift

Ce sera la conversion explicite 'as' qui sera utilisé. Le danger d'utiliser 'as' c'est que rust ne panique pas si la conversion echoue.

Par exemple :

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

Il est donc très recommandé de valider les obligations de preuve avant de traduire afin de prevenir ce genre de situation.

Extension Dans le cas où une lambda fonction non-défini dans 'b2rust_types.mch' veut etre traduit comme étant une lambda fonction de b2rust_types, il est possible d'etendre la synthaxe de typage en ajoutant des associations dans le fichier 'b2rust_operations.cfg'. Par exemple : l'utilisateur a défini bitwise_and de C dans une machine B. Il souhaite que b2rust traduit

```
    bitwise_and_uint32(aa,bb)
```

en

```
    aa ^ bb //à conversion prés
```

Il suffit qu'il ajoute la paire ' bitwise_and_uint32 and_u32' dans b2rust_operations.cfg.

Voir un exemple

3.3.2 Expression Tableau

Non terminal	Productions
ExpressionTableau	::= "{" (integer_litteral "
	| (interval)+* interval * "{" ExpressionArith "}"
	| (interval)+* interval * "{" ExpressionBooleenne "}"
	| (interval)+* interval * "{" TermeSimple "}"
	| ident

Il faut noter que integer_litteral doit etre positive.

3.4 Instructions

Les instructions B0 sont traduisible.

3.4.1 Affectation

Lors d'une affectation de la forme :

```
1  ident := Expression
```

b2rust ajoute automatiquement des conversions avec verification'try_into(_).unwrap()' pour convertir le type de l'expression vers le type de ident.
Cette conversion est utile dans le cas où l'expression a un type **compatible** mais non identique à l'ident.

Exemple en B :

```
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 – Op type

Et sa traduction en 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 Op type

Exception : Il n'est pas possible de faire des conversions concernant les types **tableaux** (mais element des tableaux c'est accepté)

Si la conversion echoue, rust panique à l'execution. Pour eviter cette situation, des lambdas fonctions fit (identité sur un domaine) sont proposés, lors de chaque affectation il est recommandé d'utilisé fit pour assurer que l'affectant a un type compatible avec l'affectable.

b2rust ignore les lambdas function fit , par exemple :

```
ident := fit_i8(Expression)
```

est traduit en

```
ident = Expression //à conversion prés
```

La synthaxe du fit peut être également étendu, comme les opérations. Voir comment etendre la synthaxe des opérations

3.4.2 Appel de fonction

Pas de contraintes en particulier, il s'agit juste d'expliquer comment l'appel de fonction est traduit. L'idée est simple:
1) copier les parametres d'entrée et de sortie

2) faire un appel de fonction sur ces copies

3) modifier les parametres de sorties avec la copie modifiée

Il s'agit d'un mécanisme pour contourner le problème d'emprunt en 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 – Exemple d'appel de fonction

Et sa traduction en 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 – Exemple d'appel de fonction

3.5 Machine Referencées

Non terminal	Productions
Clause_imports	::= “IMPORTS” ((Ident_ren “[”(” Instanciation +”,”)]”)+)

Non terminal	Productions
Clause_sees	::= “SEES” Ident_ren+
Clause_extends	::= EXTENDS” (Ident [“(” Instanciation +“,”)]+“,”)+“,” “)”])

Toute les machines références deviennent les champs du struct. A savoir les machines **importés**, **vus** et **etendus**.

Exemple en B :

```
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 – Import

Et sa traduction en rust :

```rust
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 Import

3.5.1 Parametres formelles

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

Les parametres formelles de la machine doivent être **typé** dans la clause **INVARIANT**. Le renommage

est accepté à condition de n'avoir qu'un préfixe de renommage.

Dans cette version d'atelierB, il y'a deux problemes :
- Il y'a encore des bugs sur les multiples préfixes de renommage d'atelierB.
- La clause SEES dans atelierB ne peut pas faire vraiment référence à une instance d'importation

Le choix de traduction adopté pour les parametres formelles est d'ajouter un **champ privé** dans le struct Rust.
Puis ajouter un constructeur nommé **new** en plus du constructeur par défaut.
L'instanciation des machines ayant des parametres se fera avec new.

Exemple en B :

```
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 – Import

Et sa traduction en 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 Import

3.5.2 Module de base

Concernant les machines qui n'ont pas d'implementation, b2rust génere un fichier d'extension '.rs.template' qui servira de **modèle**.

Dans le contenue du modèle :

- Les concrete_variables parfois traduit, les concrete_constants parfois traduit et toujours commenté.
- Les instructions traduisible dans l'initialisation sont parfois traduit
- La signature des opérations sont **traduite**, mais le corps de la méthode est suivi d'un macro **unimplemented!**

```
MACHINE
    error

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

```
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

Un fichier bash **check.sh** est fourni pour verifier que l'utilisateur a bien implementé le struct et les méthodes associées.

3.5.3 Fichier non destiné à être traduit

Il peut exister des machines B dont le seul but est de servir de **bibliothèque** pour fournir des informations de typage et lambdas fonctions.
Ces machines là n'ont pas vocation à etre traduit mais à générer des obligations de preuves, comme par exemple le 'b2rust_types.mch'.
Pour gérer ce type de fichier, b2rust fournit un fichier de configuration 'b2rust_exceptions.cfg', les machines se trouvant à l'intérieur de fichier ne sera pas **vu** dans la traduction par les autres machines (ne devient pas des champs du struct).
Cependant chaque type référencé de la bibliotheque qui doivent etre traduit doivent avoir une association dans b2rust_types.cfg Chaque lambdas fonctions utilisés doivent avoir une association dans b2rust_operations.cfg

Exemple en 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

```
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

Et sa traduction en rust :

```rust
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

Un Warning sera déclenché si b2rust ne reconnait pas l'information de typage d'une donnée concrete.

4 Conclusion

Pour que b2rust génère du code, il faut se poser les questions : - Est ce que tout les concrete_constants, concrete_variables sont bien typés dans la bonne clause ? - Est ce que toute les opérations ont leurs parametres d'entrée et sortie bien typés dans la précondition de la machine abstraite ? - Est ce que les variables locales sont typés avec des BecomesIn ou BecomesSuchThat ? - Est ce que les associations entre mes types et mes lambdas fonction avec ceux de b2rust_types sont bien fait ?