rust-verification

pan

May 4, 2020

Contents

theory *Definitions*
  imports *Main HOL−Library.LaTeXsugar*
begin

We present an imperative programming language with unique references called Unique, aiming at modeling the semantics of mutable references of Rust. In Rust, (mutable) references borrow the ownership, the capacity to observe and modify the content they are referring to, for a certain amount of time. References can borrow not only from a local variable or a heap allocation but also from another reference. The latter case is called reborrowing. Reborrow forms a tree-like relation between references. While the borrow checker of the Rust compiler enforces the reborrow relation to be well-formed (mutable xor alias!) statically, Unique tracks it dynamically. Imagine running an abstract interpreter of Rust with a dynamic borrow checker. At every point of use of unique references, Unique asserts that the reference is in the reborrow relation (the reference is valid) and removes every other reference reborrowed from it so that it is the only valid unique reference to the location. We expect that this dynamic nature of Unique will help us deal with (type-) unsafe portion of Rust. Note that this project is greatly influenced by R. Jung's Stacked Borrows[1].

1 Definitions and Notations

1.1 Basic data types

type-synonym *vname* = *string*
datatype *ty* = *Tbool* | *Tint* | *Tref ty*
type-synonym *tag* = *nat*
type-synonym *address* = *nat*

References consists of the address of the referent and the tag. A unique tag is assigned to each reference on creation.

datatype *val* = *VBool bool* | *VInt int* | *Reference address tag*

1.2 Expressions

datatype
  exp = *Const val*
  | *Var vname*
  | *Box exp*
  | *Reborrow exp*
  | *Deref exp*
  | *Plus exp exp*
  | *Less exp exp*
  | *Not exp*
  | *And exp exp*

Box :: *exp* ⇒ *exp* allocates memory in the heap, initializing with the argument (`Box::new` in Rust). *Box* returns a new unique reference. *Reborrow* :: *exp* ⇒ *exp* creates a new unique reference reborrowing from the argument (`&mut *p` in Rust, although most of the reborrows are automatically inserted by Rust). *Deref* :: *exp* ⇒ *exp* retrieves the content from the heap the reference pointing to (`*p` in Rust).

1.3 Commands

datatype
  com = *Skip*
  | *VarAssign vname exp* (- ::= - [*1000*, *61*] *61*)
  | *HeapAssign exp exp* (∗- ::= - [*1000*, *61*] *61*)
  | *Seq com com* (-;;/ - [*60*, *61*] *60*)
  | *If exp com com* ((*IF* -/ *THEN* -/ *ELSE* -) [*0*, *0*, *61*] *61*)
  | *WHILE exp com* ((*WHILE* -/ *DO* -) [*0*, *61*] *61*)

type-synonym *gamma* = *vname* ⇒ *val option*
type-synonym *borrow-list* = *tag list*
type-synonym *heap* = (*val* ∗ *borrow-list*) *list*

fun *kill* :: *tag* ⇒ *borrow-list* ⇒ *borrow-list* **where**
  kill t [] = [] |

  $\mathit{kill}\ t\ (x\ \#\ \mathit{xs}) = (\mathit{if}\ t = x\ \mathit{then}\ [x]\ \mathit{else}$
    $\mathit{case}\ \mathit{kill}\ t\ \mathit{xs}\ \mathit{of}$
      $[] \Rightarrow []\ |$
      $\mathit{xs} \Rightarrow x\ \#\ \mathit{xs})$

The function *kill* calculates references to be invalidated by the use of the given reference. The following two lemmas show that the used reference will be the "leaf" of the borrow tree.

lemma $[\mathit{simp}]$: $t \notin \mathit{set}\ \mathit{ts} \Longrightarrow \mathit{kill}\ t\ \mathit{ts} = []$
$\langle \mathit{proof} \rangle$

lemma $t \in \mathit{set}\ \mathit{ts} \Longrightarrow \exists\, \mathit{ts}'.\ \mathit{kill}\ t\ \mathit{ts} = \mathit{ts}'\ @\ [t]$
$\langle \mathit{proof} \rangle$

fun $\mathit{tags} :: \mathit{heap} \Rightarrow \mathit{tag}\ \mathit{list}$ **where**
  $\mathit{tags}\ H = \mathit{fold}\ (\lambda e\ \mathit{ts}.\ \mathit{ts}\ @\ (\mathit{snd}\ e))\ H\ []$

fun $\mathit{fresh\text{-}tag} :: \mathit{heap} \Rightarrow \mathit{tag}$ **where**
  $\mathit{fresh\text{-}tag}\ H = \mathit{fold}\ \mathit{max}\ (\mathit{tags}\ H)\ 0 + 1$

Function $\mathit{fresh\text{-}tag} :: (\mathit{val} \times \mathit{nat}\ \mathit{list})\ \mathit{list} \Rightarrow \mathit{nat}$ generates a new tag which doesn't exist in the tree. (FIXME: I figure that *fresh-tag* may reuse tags if they have been already killed. We need to record all the tags generated.)

fun $\mathit{writable} :: \mathit{address} \Rightarrow \mathit{tag} \Rightarrow \mathit{heap} \Rightarrow \mathit{bool}$ **where**
  $\mathit{writable}\ p\ t\ H \longleftrightarrow p < \mathit{length}\ H \wedge t \in \mathit{set}\ (\mathit{snd}\ (H\ !\ p))$
definition *readable* **where** $\mathit{readable} = \mathit{writable}$

Functions $\mathit{writable} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow (\mathit{val} \times \mathit{nat}\ \mathit{list})\ \mathit{list} \Rightarrow \mathit{bool}$ and *readable* $:: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow (\mathit{val} \times \mathit{nat}\ \mathit{list})\ \mathit{list} \Rightarrow \mathit{bool}$ determine whether the given reference can be used to write to/read from it. The validity of references is determined by the existence in the current borrow tree. In Unique, we allow only unique references. Thus *readable* is equivalent to *writable*.

end
theory *Exp*
  imports *Definitions*
begin

2 Semantics of Expressions

This section presents the big-step semantics of expressions.

inductive $\mathit{exp\text{-}sem} ::$
  $\mathit{gamma} * \mathit{heap} * \mathit{exp} \Rightarrow \mathit{heap} * \mathit{val} \Rightarrow \mathit{bool}$ (**infix** $\Downarrow$ *65*)
where
  Const: $(\Gamma,\ H,\ \mathit{Const}\ v) \Downarrow (H,\ v)\ |$
  Var: $\mathit{the}\ (\Gamma\ x) = v \Longrightarrow (\Gamma,\ H,\ \mathit{Var}\ x) \Downarrow (H,\ v)\ |$
  Box: $[\![(\Gamma,\ H,\ e) \Downarrow (H',\ v);\ p = \mathit{length}\ H';\ t = \mathit{fresh\text{-}tag}\ H']\!]$

$\Longrightarrow (\Gamma,\ H,\ \mathit{Box}\ e) \Downarrow (H'\ @\ [(v,\ [t])],\ \mathit{Reference}\ p\ t)$ |
Reborrow: $[\![(\Gamma,\ H,\ e) \Downarrow (H',\ \mathit{Reference}\ p\ t);\ \mathit{writable}\ p\ t\ H';\ (v,\ ts) = H'\ !\ p;\ t' = \mathit{fresh\text{-}tag}\ H']\!]$
$\Longrightarrow (\Gamma,\ H,\ \mathit{Reborrow}\ e) \Downarrow (H'[p := (v,\ (\mathit{kill}\ t\ ts)\ @\ [t'])],\ \mathit{Reference}\ p\ t')$ |
Deref: $[\![(\Gamma,\ H,\ e) \Downarrow (H',\ \mathit{Reference}\ p\ t);\ \mathit{readable}\ p\ t\ H';\ (v,\ ts) = H'\ !\ p]\!]$
$\Longrightarrow (\Gamma,\ H,\ \mathit{Deref}\ e) \Downarrow (H'[p := (v,\ \mathit{kill}\ t\ ts)],\ v)$ |
Plus: $[\![(\Gamma,\ H,\ e1) \Downarrow (H',\ \mathit{VInt}\ i1);\ (\Gamma,\ H',\ e2) \Downarrow (H'',\ \mathit{VInt}\ i2)]\!]$
$\Longrightarrow (\Gamma,\ H,\ \mathit{Plus}\ e1\ e2) \Downarrow (H'',\ \mathit{VInt}\ (i1 + i2))$ |
Less: $[\![(\Gamma,\ H,\ e1) \Downarrow (H',\ \mathit{VInt}\ i1);\ (\Gamma,\ H',\ e2) \Downarrow (H'',\ \mathit{VInt}\ i2)]\!]$
$\Longrightarrow (\Gamma,\ H,\ \mathit{Less}\ e1\ e2) \Downarrow (H'',\ \mathit{VBool}\ (i1 < i2))$ |
Not: $(\Gamma,\ H,\ e) \Downarrow (H',\ \mathit{VBool}\ b) \Longrightarrow (\Gamma,\ H,\ \mathit{Not}\ e) \Downarrow (H',\ \mathit{VBool}\ (\neg b))$ |
And: $[\![(\Gamma,\ H,\ e1) \Downarrow (H',\ \mathit{VBool}\ b1);\ (\Gamma,\ H',\ e2) \Downarrow (H'',\ \mathit{VBool}\ b2)]\!]$
$\Longrightarrow (\Gamma,\ H,\ \mathit{And}\ e1\ e2) \Downarrow (H'',\ \mathit{VBool}\ (b1 \wedge b2))$

We present the intuition behind the selected rules. First, Box rule allocates a new slot at the end of heap, returning a reference pointing to the slot with a fresh tag.

$$\frac{(\Gamma,\ H,\ e) \Downarrow (H',\ v) \qquad p = |H'| \qquad t = \mathit{fresh\text{-}tag}\ H'}{(\Gamma,\ H,\ \mathit{Box}\ e) \Downarrow (H'\ @\ [(v,\ [t])],\ \mathit{Reference}\ p\ t)}\ \textsc{Box}$$

Next, Deref rule allows us to access the content the given reference is pointing to. The validity of the reference is checked by $\mathit{readable} :: \mathit{nat} \Rightarrow \mathit{nat} \Rightarrow (\mathit{val} \times \mathit{nat\ list})\ \mathit{list} \Rightarrow \mathit{bool}$. Moreover, Deref invalidates all descendant references to establish the uniqueness ($\mathit{kill}\ t\ ts$).

$$\frac{\begin{array}{c}(\Gamma,\ H,\ e) \Downarrow (H',\ \mathit{Reference}\ p\ t)\\ \mathit{readable}\ p\ t\ H' \qquad (v,\ ts) = H'_{[p]}\end{array}}{(\Gamma,\ H,\ \mathit{Deref}\ e) \Downarrow (H'[p := (v,\ \mathit{kill}\ t\ ts)],\ v)}\ \textsc{Deref}$$

Last but not least, Reborrow rule creates a new reference with a fresh tag reborrowing the given reference. The newly created reference is writable. Thus the original reference must be valid for writes. Reborrow is considered as the use of the original reference and hence it invalidates former children of the reference. As a result, the reborrow tree will have the original reference and the reborrowing reference as the last two element.

Reborrow rule:

$$\frac{\begin{array}{c}(\Gamma,\ H,\ e) \Downarrow (H',\ \mathit{Reference}\ p\ t)\\ \mathit{writable}\ p\ t\ H' \qquad (v,\ ts) = H'_{[p]} \qquad t' = \mathit{fresh\text{-}tag}\ H'\end{array}}{(\Gamma,\ H,\ \mathit{Reborrow}\ e) \Downarrow (H'[p := (v,\ \mathit{kill}\ t\ ts\ @\ [t'])],\ \mathit{Reference}\ p\ t')}$$

We can prove that the semantics of expressions is deterministic.

lemma *exp-sem-deterministic*: $E \Downarrow E' \Longrightarrow E \Downarrow E'' \Longrightarrow E'' = E'$
⟨*proof*⟩

The following lemmas are a sanity check of the semantics. We must be able to write through the references retrieved by Box and Reborrow

lemma *box--writable*: $(\Gamma,\ H,\ \mathit{Box}\ e) \Downarrow (H',\ \mathit{Reference}\ p\ t) \Longrightarrow \mathit{writable}\ p\ t\ H'$
  ⟨*proof*⟩

lemma *reborrow--writable*: $(\Gamma,\ H,\ \mathit{Reborrow}\ e) \Downarrow (H',\ \mathit{Reference}\ p\ t) \Longrightarrow \mathit{writable}$ $p\ t\ H'$
  ⟨*proof*⟩

code-pred [*show-modes*] *exp-sem* ⟨*proof*⟩

end
theory *Unique*
  imports *Main Definitions Exp Star*
begin

3 Semantics of Commands

This section describes the small-step semantics of commands. The configuration of an execution consists of

- the variable environment Γ (of type *gamma*);
- the heap H (of type *heap*) which holds the actual data and the reborrow trees for each address; and
- the command to be executed c (of type *com*).

type-synonym *config* = *gamma* ∗ *heap* ∗ *com*
inductive *com-sem* :: *config* ⇒ *config* ⇒ *bool* (**infix** → *65*)
where
  VAssign: $(\Gamma,\ H,\ e) \Downarrow (H',\ v)$
      $\Longrightarrow (\Gamma,\ H,\ x ::= e) \rightarrow (\Gamma(x \mapsto v),\ H',\ \mathit{Skip})$ |
  HAssign: $[\![(\Gamma,\ H,\ e_l) \Downarrow (H',\ \mathit{Reference}\ p\ t);$
      $(\Gamma,\ H',\ e_v) \Downarrow (H'',\ v);$
      $\mathit{writable}\ p\ t\ H'']\!]$
      $\Longrightarrow (\Gamma,\ H,\ {*e_l} ::= e_v) \rightarrow (\Gamma,\ H''[p := (v,\ \mathit{kill}\ t\ (\mathit{snd}\ (H''\ !\ p)))],\ \mathit{Skip})$ |
  SeqL: $(\Gamma,\ H,\ \mathit{Skip};;\ c) \rightarrow (\Gamma,\ H,\ c)$ |
  SeqR: $(\Gamma,\ H,\ c1) \rightarrow (\Gamma',\ H',\ c') \Longrightarrow (\Gamma,\ H,\ c1;;\ c2) \rightarrow (\Gamma',\ H',\ c';;\ c2)$ |
  IfTrue: $(\Gamma,\ H,\ b) \Downarrow (H',\ \mathit{VBool}\ \mathit{True})$
      $\Longrightarrow (\Gamma,\ H,\ \mathit{IF}\ b\ \mathit{THEN}\ c1\ \mathit{ELSE}\ c2) \rightarrow (\Gamma,\ H',\ c1)$ |
  IfFlase: $(\Gamma,\ H,\ b) \Downarrow (H',\ \mathit{VBool}\ \mathit{False})$
      $\Longrightarrow (\Gamma,\ H,\ \mathit{IF}\ b\ \mathit{THEN}\ c1\ \mathit{ELSE}\ c2) \rightarrow (\Gamma,\ H',\ c2)$ |

While: $(\Gamma,\ H,\ \textit{WHILE b DO c}) \to (\Gamma,\ H,\ \textit{IF b THEN } (c;;\ \textit{WHILE b DO c})\ \textit{ELSE Skip})$

code-pred [*show-modes*] *com-sem* ⟨*proof*⟩

abbreviation *com-sem-steps* :: *config* ⇒ *config* ⇒ *bool* (**infix** $\to^*$ *65*) **where**
  $cfg \to^* cfg' == \textit{star com-sem } cfg\ cfg'$

Figure 1 shows the assignment rules. Other commands are standard.

The VAssign rule assigns a value to the variable while the HAssign rule to the location the reference on the left hand side is referring to. To assign through a reference, the reference must be valid for writes. Moreover, the write is considered as a use of the reference; it will invalidate the child references.

$$\frac{(\Gamma,\ H,\ e) \Downarrow (H',\ v)}{(\Gamma,\ H,\ x ::= e) \to (\Gamma(x \mapsto v),\ H',\ \textit{Skip})}\ \textsc{VAssign}$$

$$\frac{(\Gamma,\ H,\ e_l) \Downarrow (H',\ \textit{Reference } p\ t) \qquad (\Gamma,\ H',\ e_v) \Downarrow (H'',\ v) \qquad \textit{writable } p\ t\ H''}{(\Gamma,\ H,\ *e_l ::= e_v) \to (\Gamma,\ H''[p := (v,\ \textit{kill } t\ (\textit{snd } H''_{[p]}))],\ \textit{Skip})}\ \textsc{HAssign}$$

Figure 1: Assignment rules

The following lemmas show that the execution of the commands is deterministic.

definition *final* :: *config* ⇒ *bool* **where**
  final cfg ⟷ (*case cfg of* (-, -, *Skip*) ⇒ *True* | - ⇒ *False*)

definition *stuck* :: *config* ⇒ *bool* **where**
  stuck cfg ⟷ (*case cfg of*
    (-, -, *Skip*) ⇒ *False* |
    - ⇒ $\neg(\exists\, cfg'.\ cfg \to cfg')$)

lemma *skip-final-noteq*[*simp*]: $(\Gamma,\ H,\ c) \to (\Gamma',\ H',\ c') \Longrightarrow c \neq \textit{Skip}$
⟨*proof*⟩

lemma *skip-final*[*simp*]: $\neg(\Gamma,\ H,\ \textit{Skip}) \to (\Gamma',\ H',\ c)$
⟨*proof*⟩

lemma *com-sem-deterministic*: $cfg \to cfg' \Longrightarrow cfg \to cfg'' \Longrightarrow cfg'' = cfg'$
⟨*proof*⟩

We show the transitivity of $(\rightarrow^*)$.

lemma *com-seql-trans*:
 $(\Gamma,\ H,\ c1) \rightarrow^* (\Gamma',\ H',\ c1') \Longrightarrow (\Gamma,\ H,\ c1;;\ c2) \rightarrow^* (\Gamma',\ H',\ c1';;\ c2)$
$\langle proof\rangle$

4 Code examples

4.1 Invalidating a reference

Consider the following Rust program.

```rust
1 let mut root = Box::new(42);
2 let mut px = &mut *root; // reborrowing from root
3 *px = 100;
4 let mut py = &mut *root; // another reborrow. invalidates px
5 *py = 200;
6 // *px = 300 // this write is invalid.
```

In line 1, we create a box that contains 42 in the heap. In the following two lines, we reborrow from it and use the reference to write. The interesting part is line 4. In this line, we create a new reference from `root`. As this reborrow asserts that `py` is the unique reference to the box, `px` must be invalidated at this point. Therefore, if we remove the comment in line 6, the program will perform an invalid write (thus rustc will reject the program).

Let's execute the program with Unique. The following command is the translation of the program above without the last commented line.

definition XY :: *com* **where**
 $XY =$
  $''root'' ::= Box\ (Const\ (VInt\ 42));;$
  $''px'' ::= Reborrow\ (Var\ ''root'');;$
  $*(Var\ ''px'') ::= Const\ (VInt\ 100);;$
  $''py'' ::= Reborrow\ (Var\ ''root'');;$
  $*(Var\ ''py'') ::= Const\ (VInt\ 200)$

We can let Isabelle compute the program.

values $\{(map\ \Gamma\ [''root'',\ ''px'',\ ''py''],\ H,\ c) \mid$
  $\Gamma\ H\ c.\ (Map.empty,\ [],\ XY) \rightarrow^* (\Gamma,\ H,\ c)\}$

The following shows the variable environment and the heap at the end of execution.

abbreviation Γ_{XY} :: *gamma* **where**
 $\Gamma_{XY} == [''root'' \mapsto Reference\ 0\ 1,\ ''px'' \mapsto Reference\ 0\ 2,\ ''py'' \mapsto Reference\ 0\ 3]$
abbreviation H_{XY} :: *heap* **where**
 $H_{XY} == [(VInt\ 200,\ [1,\ 3])]$

We can also prove that the program result in the state shown above in theory, but it's really tedious (why can't Isabelle prove it by computing the execution?). The actual proof is left to the reader.

lemma *final-xy*: $(\mathit{Map.empty},\ [],\ XY) \rightarrow^* (\Gamma_{XY},\ H_{XY},\ \mathit{Skip})$
  sorry

We show that accessing through `px` after the program will stuck.

lemma *intermediate-xyx*: $(\mathit{Map.empty},\ [],\ XY;;\ *(\mathit{Var}\ ''px'')\ ::=\ \mathit{Const}\ (\mathit{VInt}\ 300)) \rightarrow^*$
  $(\Gamma_{XY},\ H_{XY},\ *(\mathit{Var}\ ''px'')\ ::=\ \mathit{Const}\ (\mathit{VInt}\ 300))$
  $\langle \mathit{proof} \rangle$

lemma *stuck-xyx*: *stuck*
  $(\Gamma_{XY},\ H_{XY},\ *(\mathit{Var}\ ''px'')\ ::=\ \mathit{Const}\ (\mathit{VInt}\ 300))$
  $\langle \mathit{proof} \rangle$

lemma $\exists\, \mathit{cfg}.\ (\mathit{Map.empty},\ [],\ XY;;\ *(\mathit{Var}\ ''px'')\ ::=\ \mathit{Const}\ (\mathit{VInt}\ 300)) \rightarrow^* \mathit{cfg}$
  $\wedge\ \mathit{stuck}\ \mathit{cfg}$
  $\langle \mathit{proof} \rangle$

4.2 Swap

The following program swaps the content the given two references pointing to.

```
fun swap :: exp ⇒ exp ⇒ com where
  swap x y =
    ''reborrow-x'' ::= Reborrow x;;
    ''reborrow-y'' ::= Reborrow y;;
    ''tmp'' ::= Deref (Var ''reborrow-x'');;
    *(Var ''reborrow-x'') ::= Deref (Var ''reborrow-y'');;
    *(Var ''reborrow-y'') ::= Var ''tmp''
```

We can see the content swapped during the execution by running it on Isabelle.

values $\{H \mid \Gamma\ H\ c.\ (\mathit{Map.empty},\ [],\ \mathit{swap}\ (\mathit{Box}\ (\mathit{Const}\ (\mathit{VInt}\ 42)))\ (\mathit{Box}\ (\mathit{Const}\ (\mathit{VInt}\ 100)))) \rightarrow^* (\Gamma,\ H,\ c)\}$

Proving the correctness of *swap* would need a Hoare Logic. The road continues...

end

References

[1] R. Jung, H. Dang, J. Kang, and D. Dreyer. Stacked borrows: an aliasing model for rust. *Proc. ACM Program. Lang.*, 4(POPL):41:1–41:32, 2020.