rust-verification

pan

September 3, 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*
datatype *tag-kind* = *Unique*

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 *place* = *Var vname* | *Deref place*

Datatype *place* represents an access path to data. It corresponds to `Place` in https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/struct.Place.html

datatype *operand* = *Place place* | *Constant val*

Datatype *operand* describes a value inside an rvalue. *Place* denotes the current value at the place, while *Constant* denotes a constant value. *operand* correnspونds to `Place` in https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/enum.Operand.html, though we simplified the differentiation between a move and a copy (i.e. Unique doesn't track the ownership).

datatype
rvalue = *Use operand*
| *Box operand*
| *Ref place*
| *Reborrow place*
| *Plus operand operand*
| *Less operand operand*
| *Not operand*
| *And operand operand*

Datatype *rvalue* corresponds to an expression in a usual programming language (as in the previous version of Unique). We chose the term rvalue because of parity with Rust MIR (https://doc.rust-lang.org/nightly/nightly-rustc/rustc_middle/mir/enum.Rvalue.html)

Note that *rvalue* is *not* recursive. Compound expressions need to be broken apart so that intermediate computation is stored to a variable.

The semantics of the constructs is as follows:

- *Box* :: *operand* ⇒ *rvalue* allocates memory in the heap, initializing with the argument (`Box::new` in Rust). *Box* returns a new unique reference. Note that Box in MIR doesn't initialize the location but only allocates. The semantics will be adapted to MIR's one as we formalize uninitialized memory.
- *Ref* :: *place* ⇒ *rvalue* creates a unique reference pointing to the place.
- *Reborrow* :: *place* ⇒ *rvalue* creates a new unique reference reborrowing from the argument (`&mut *p` in Rust, although most of the reborrows are automatically inserted by Rust). (TODO: I guess `Use(Copy(mutable ref))` in MIR corresponds to Reborrow in Unique.)

1.3 Commands

datatype
com = *Skip*
| *Assign place rvalue* (- ::= - [*1000*, *61*] *61*)
| *Seq com com* (-;;/ - [*60*, *61*] *60*)
| *If rvalue com com* ((*IF* -/ *THEN* -/ *ELSE* -) [*0*, *0*, *61*] *61*)
| *WHILE rvalue com* ((*WHILE* -/ *DO* -) [*0*, *61*] *61*)

type-synonym *gamma* = *vname* ⇒ *address* ∗ *tag*

Unique implicitly performs boxing to variables. In other words, every variable behaves as a root reference to a memory cell.

type-synonym *tags* = *tag-kind list*
type-synonym *borrow-list* = *tag list*
type-synonym *heap* = (*val* ∗ *borrow-list*) *list*

fun *kill* :: *tag* ⇒ *borrow-list* ⇒ *borrow-list* **where**
kill t [] = [] |
kill t (*x* # *xs*) = (*if t* = *x then* [*x*] *else*
case kill t xs of
[] ⇒ [] |
xs ⇒ *x* # *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 [*simp*]: $t \notin set\ ts \Longrightarrow kill\ t\ ts = []$
⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩
lemma $t \in set\ ts \Longrightarrow \exists ts'.\ kill\ t\ ts = ts' @ [t]$
⟨*proof*⟩

fun *kill-heap* :: (*address* ∗ *tag*) ⇒ *heap* ⇒ *heap* **where**
kill-heap (*a*, *t*) *H* = (*let* (*v*, *ts*) = *H* ! *a in H*[*a* := (*v*, *kill t ts*)])

abbreviation *kill-all* :: (*address* ∗ *tag*) *list* ⇒ *heap* ⇒ *heap* **where**
kill-all refs H == *foldr kill-heap refs H*

fun *writable* :: *heap* ⇒ (*address* ∗ *tag*) ⇒ *bool* **where**
writable H (*p*, *t*) ⟷ *p* < *length H* ∧ *t* ∈ *set* (*snd* (*H* ! *p*))
definition *readable* **where** *readable* = *writable*

Functions *writable* :: (*val* × *nat list*) *list* ⇒ *nat* × *nat* ⇒ *bool* and *readable* :: (*val* × *nat list*) *list* ⇒ *nat* × *nat* ⇒ *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*.

definition *allocated* :: *heap* ⇒ *address* ⇒ *bool* **where**
allocated H a ⟷ *a* < *length H*
⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩
end
theory *Exp*
imports *Definitions*
begin

2 Semantics of Expressions

This section presents the big-step semantics of expressions.

Introducing a locale here looks a good idea as it helps us formulate axioms among the heap access modifier. Expected axioms:

- Modifier must be idempotent: $m\ p\ (m\ p\ H) = m\ p\ H$.
- Modifier must not allocate/deallocate a memory cell: $|m\ p\ H| = |H|$.
- The content is left intact: $\mathit{fst}\ (m\ p\ H)_{[a]} = \mathit{fst}\ H_{[a]}$

type-synonym *modifier* = (*address* ∗ *tag*) ⇒ *heap* ⇒ *heap option*
fun *write-access* :: *modifier* **where**
write-access (*a*, *t*) *H* =
(*if writable H* (*a*, *t*) *then*
let (*v*, *ts*) = *H* ! *a in*
Some (*H*[*a* := (*v*, *kill t ts*)])
else None)
fun *read-access* :: *modifier* **where**
read-access (*a*, *t*) *H* =
(*if readable H* (*a*, *t*) *then*
let (*v*, *ts*) = *H* ! *a in*
Some (*H*[*a* := (*v*, *kill t ts*)])
else None)

inductive *place-sem* ::
$\quad$ *gamma* ⇒ *modifier* ⇒ *heap* ∗ *place* ⇒ *heap* ∗ *val* ⇒ *bool*
$\quad$ $((\text{-}; \text{-} \vdash \text{-} \Downarrow_p \text{-}))$ **for** Γ m
where
$\quad$ Var: $\Gamma\ x = (a,\ t) \Longrightarrow \Gamma;\ m \vdash (H,\ \mathit{Var}\ x) \Downarrow_p (H,\ \mathit{Reference}\ a\ t)$ |
$\quad$ Deref: $[\![\Gamma;\ m \vdash (H,\ p) \Downarrow_p (H',\ \mathit{Reference}\ a\ t);\ m\ (a,\ t)\ H' = \mathit{Some}\ H'']\!] \Longrightarrow$
$\qquad\quad \Gamma;\ m \vdash (H,\ \mathit{Deref}\ p) \Downarrow_p (H'',\ \mathit{fst}\ (H''\ !\ a))$

The relation *place-sem* describes the semantics of places with respect to a heap access modifier m. $\Gamma;\ m \vdash (H,\ p) \Downarrow_p (H',\ v)$ means that a place expression p evaluates to a value v under the environment Γ and H, while the access has changed the state of the heap to H'.

We give a shorthand for read and write access to the place.

abbreviation *read-place-sem* :: *gamma* ⇒ *heap* ∗ *place* ⇒ *heap* ∗ *val* ⇒ *bool*
$\quad$ $((\text{-} \vdash_r \text{-} \Downarrow_p \text{-}))$
$\quad$ **where**
$\quad$ $\Gamma \vdash_r p \Downarrow_p v == \Gamma;\ \mathit{read\text{-}access} \vdash p \Downarrow_p v$
abbreviation *write-place-sem* :: *gamma* ⇒ *heap* ∗ *place* ⇒ *heap* ∗ *val* ⇒ *bool*
$\quad$ $((\text{-} \vdash_w \text{-} \Downarrow_p \text{-}))$
$\quad$ **where**
$\quad$ $\Gamma \vdash_w p \Downarrow_p v == \Gamma;\ \mathit{write\text{-}access} \vdash p \Downarrow_p v$

Note that every operand is considered as a read access to the place.

inductive *operand-sem* ::
$\quad$ *gamma* ⇒ *heap* ∗ *operand* ⇒ *heap* ∗ *val* ⇒ *bool*
$\quad$ $((\text{-} \vdash \text{-} \Downarrow_{op} \text{-}))$ **for** Γ
where
$\quad$ Place: $\Gamma \vdash_r (H,\ p) \Downarrow_p v \Longrightarrow \Gamma \vdash (H,\ \mathit{Place}\ p) \Downarrow_{op} v$ |
$\quad$ $\mathit{Constant}$: $\Gamma \vdash (H,\ \mathit{Constant}\ v) \Downarrow_{op} (H,\ v)$

inductive *rvalue-sem* ::
$\quad$ *gamma* ⇒ *tags* ∗ *heap* ∗ *rvalue* ⇒ *tags* ∗ *heap* ∗ *val* ⇒ *bool*
$\quad$ $((\text{-} \vdash \text{-} \Downarrow \text{-}))$ **for** Γ
where
$\quad$ Use: $\Gamma \vdash (H,\ op) \Downarrow_{op} (H',\ v) \Longrightarrow \Gamma \vdash (T,\ H,\ \mathit{Use}\ op) \Downarrow (T,\ H',\ v)$ |
$\quad$ Box: $[\![\Gamma \vdash (H,\ op) \Downarrow_{op} (H',\ v);\ p = \mathit{length}\ H';\ t = \mathit{length}\ T]\!]$
$\qquad \Longrightarrow \Gamma \vdash (T,\ H,\ \mathit{Box}\ op) \Downarrow (T\ @\ [\mathit{Unique}],\ H'\ @\ [(v,\ [t])],\ \mathit{Reference}\ p\ t)$ |
$\quad$ Ref: $\Gamma \vdash_w (H,\ p) \Downarrow_p (H',\ \mathit{Reference}\ a\ t) \Longrightarrow \Gamma \vdash (T,\ H,\ \mathit{Ref}\ p) \Downarrow (T,\ H',$ $\mathit{Reference}\ a\ t)$ |
$\quad$ $\mathit{Reborrow}$: $[\![\Gamma \vdash_w (H,\ p) \Downarrow_p (H',\ \mathit{Reference}\ a\ t);\ \mathit{writable}\ H\ (a,\ t);$
$\qquad\qquad t' = \mathit{length}\ T;\ H'\ !\ a = (v,\ ts)]\!]$
$\qquad\qquad \Longrightarrow \Gamma \vdash (T,\ H,\ \mathit{Reborrow}\ p) \Downarrow (T\ @\ [\mathit{Unique}],\ H'[a := (v,\ ts\ @\ [t'])],$ $\mathit{Reference}\ a\ t')$ |
$\quad$ Plus: $[\![\Gamma \vdash (H,\ \mathit{lhs}) \Downarrow_{op} (H',\ \mathit{VInt}\ \mathit{lhs}');\ \Gamma \vdash (H',\ \mathit{rhs}) \Downarrow_{op} (H'',\ \mathit{VInt}\ \mathit{rhs}')]\!]$
$\qquad \Longrightarrow \Gamma \vdash (T,\ H,\ \mathit{Plus}\ \mathit{lhs}\ \mathit{rhs}) \Downarrow (T,\ H'',\ \mathit{VInt}\ (\mathit{lhs}' + \mathit{rhs}'))$ |
$\quad$ Less: $[\![\Gamma \vdash (H,\ \mathit{lhs}) \Downarrow_{op} (H',\ \mathit{VInt}\ \mathit{lhs}');\ \Gamma \vdash (H',\ \mathit{rhs}) \Downarrow_{op} (H'',\ \mathit{VInt}\ \mathit{rhs}')]\!]$
$\qquad \Longrightarrow \Gamma \vdash (T,\ H,\ \mathit{Less}\ \mathit{lhs}\ \mathit{rhs}) \Downarrow (T,\ H'',\ \mathit{VBool}\ (\mathit{lhs}' < \mathit{rhs}'))$ |

Not: $\Gamma \vdash (H,\ op) \Downarrow_{op} (H',\ VBool\ v) \Longrightarrow \Gamma \vdash (T,\ H,\ Not\ op) \Downarrow (T,\ H',\ VBool\ (\neg v))$ |
And: $[\![\Gamma \vdash (H,\ lhs) \Downarrow_{op} (H',\ VBool\ lhs'); \Gamma \vdash (H',\ rhs) \Downarrow_{op} (H'',\ VBool\ rhs')]\!]$
$\Longrightarrow \Gamma \vdash (T,\ H,\ And\ lhs\ rhs) \Downarrow (T,\ H'',\ VBool\ (lhs' \wedge rhs'))$

Some TODO notes on the semantics of expressions:

- I don't get the semantics of *Ref* in MIR and how it should behave under the auto-boxing of variables.
- I'm not sure readable/writable checking is correct.

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.

$$\frac{\Gamma \vdash (H,\ op) \Downarrow_{op} (H',\ v) \qquad p = |H'| \qquad t = |T|}{\Gamma \vdash (T,\ H,\ Box\ op) \Downarrow (T\ @\ [Unique],\ H'\ @\ [(v,\ [t])],\ Reference\ p\ t)}\ \textsc{Box}$$

Next, Use and Ref rules allow us to read the content through a place (or give a constant).

$$\frac{\Gamma \vdash (H,\ op) \Downarrow_{op} (H',\ v)}{\Gamma \vdash (T,\ H,\ Use\ op) \Downarrow (T,\ H',\ v)}\ \textsc{Use}$$

$$\frac{\Gamma \vdash_w (H,\ p) \Downarrow_p (H',\ Reference\ a\ t)}{\Gamma \vdash (T,\ H,\ Ref\ p) \Downarrow (T,\ H',\ Reference\ a\ t)}\ \textsc{Ref}$$

Last but not least, Reborrow rule creates a new reference pointing to the same object but with a fresh tag. 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:

$$\frac{\Gamma \vdash_w (H,\ p) \Downarrow_p (H',\ Reference\ a\ t) \qquad writable\ H\ (a,\ t) \qquad t' = |T| \qquad H'_{[a]} = (v,\ ts)}{\Gamma \vdash (T,\ H,\ Reborrow\ p) \Downarrow (T\ @\ [Unique],\ H'[a := (v,\ ts\ @\ [t'])],\ Reference\ a\ t')}$$

We can prove that the semantics of expressions is deterministic.

lemma *place-sem-det*: $\Gamma;\ m \vdash p \Downarrow_p v \Longrightarrow \Gamma;\ m \vdash p \Downarrow_p v' \Longrightarrow v' = v$
⟨*proof*⟩
lemma *place-sem-det'*: $\Gamma;\ m \vdash (H,\ p) \Downarrow_p (H',\ v') \Longrightarrow \Gamma;\ m \vdash (H,\ p) \Downarrow_p (H'',\ v'')$
$\Longrightarrow (H'',\ v'') = (H',\ v')$
⟨*proof*⟩

lemma *operand-sem-det*: $\Gamma \vdash op \Downarrow_{op} v \Longrightarrow \Gamma \vdash op \Downarrow_{op} v' \Longrightarrow v' = v$
⟨*proof*⟩
lemma *operand-sem-det'*: $\Gamma \vdash (H,\ op) \Downarrow_{op} (H',\ v') \Longrightarrow \Gamma \vdash (H,\ op) \Downarrow_{op} (H'',\ v'')$
$\Longrightarrow (H'',\ v'') = (H',\ v')$
⟨*proof*⟩

lemma *rvalue-sem-det*: $\Gamma \vdash (T,\ H,\ rv) \Downarrow (T',\ H',\ v') \Longrightarrow \Gamma \vdash (T,\ H,\ rv) \Downarrow (T'',$
$H'',\ v'')$
$\Longrightarrow (T'',\ H'',\ v'') = (T',\ H',\ v')$
⟨*proof*⟩

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

lemma *box-writable*: $\Gamma \vdash (T,\ H,\ Box\ e) \Downarrow (T',\ H',\ Reference\ a\ t) \Longrightarrow writable$
$H'\ (a,\ t)$
⟨*proof*⟩

lemma *wa-preserve-length*: $write\text{-}access\ at\ H = Some\ H' \Longrightarrow length\ H' = length$
H
⟨*proof*⟩

lemma *write-preserve-length*: $\Gamma \vdash_w p \Downarrow_p v \Longrightarrow length\ (fst\ p) = length\ (fst\ v)$
⟨*proof*⟩

lemma *reborrow-writable*: $\Gamma \vdash (T,\ H,\ Reborrow\ p) \Downarrow (T',\ H',\ Reference\ a\ t) \Longrightarrow$
$writable\ H'\ (a,\ t)$
⟨*proof*⟩

end
⟨*proof*⟩⟨*proof*⟩
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 tags issued T (of type *tags*);

- 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*). The variable environment Γ (of type *gamma*) is fixed throughout the execution.

type-synonym *config* = *tags* ∗ *heap* ∗ *com*
inductive *com-sem* :: *gamma* ⇒ *config* ⇒ *config* ⇒ *bool* ((- ⊢ - → -)) **for** Γ
where
Assign: $[\![\Gamma \vdash_w (H,\ p) \Downarrow_p (H',\ Reference\ a\ t);\ writable\ H'\ (a,\ t);\ H'' = kill\text{-}heap\ (a,\ t)\ H';$
$\Gamma \vdash (T,\ H'',\ rv) \Downarrow (T',\ H''',\ v);\ H'''\ !\ a = (\text{-},\ ts)]\!]$
$\Longrightarrow \Gamma \vdash (T,\ H,\ p ::= rv) \to (T',\ H''[a := (v,\ ts)],\ Skip)$ |
SeqL: $\Gamma \vdash (T,\ H,\ Skip;;\ c) \to (T,\ H,\ c)$ |
SeqR: $\Gamma \vdash (T,\ H,\ c1) \to (T',\ H',\ c') \Longrightarrow \Gamma \vdash (T,\ H,\ c1;;\ c2) \to (T',\ H',\ c';;\ c2)$ |
IfTrue: $\Gamma \vdash (T,\ H,\ b) \Downarrow (T',\ H',\ VBool\ True)$
$\Longrightarrow \Gamma \vdash (T,\ H,\ IF\ b\ THEN\ c1\ ELSE\ c2) \to (T',\ H',\ c1)$ |
IfFalse: $\Gamma \vdash (T,\ H,\ b) \Downarrow (T',\ H',\ VBool\ False)$
$\Longrightarrow \Gamma \vdash (T,\ H,\ IF\ b\ THEN\ c1\ ELSE\ c2) \to (T',\ H',\ c2)$ |
While: $\Gamma \vdash (T,\ H,\ WHILE\ b\ DO\ c) \to (T,\ H,\ IF\ b\ THEN\ (c;;\ WHILE\ b\ DO\ c)\ ELSE\ Skip)$

abbreviation *com-sem-steps* :: *gamma* ⇒ *config* ⇒ *config* ⇒ *bool* ((- ⊢ - →* -))
where
$\Gamma \vdash cfg \to^* cfg' == star\ (com\text{-}sem\ \Gamma)\ cfg\ cfg'$

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

The Assign rule assigns a value to the the location. First, $\Gamma \vdash_w (H,\ p) \Downarrow_p (H',\ Reference\ a\ t)$ computes the location with write modifier. Next, $writable\ H'\ (a,\ t)$ and $H'' = kill\text{-}heap\ (a,\ t)\ H'$ assert that the reference is writable and unique. Note that this assertion is made *before* the evaluation of the right hand side. This means Unique does not support Two Phase Borrows currently. Finally, we compute the right hand side and assigns the result to the location.

$$\frac{\begin{array}{c}\Gamma \vdash_w (H,\ p) \Downarrow_p (H',\ Reference\ a\ t) \\ writable\ H'\ (a,\ t) \qquad H'' = kill\text{-}heap\ (a,\ t)\ H' \\ \Gamma \vdash (T,\ H'',\ rv) \Downarrow (T',\ H''',\ v) \qquad H'''_{[a]} = (uu,\ ts)\end{array}}{\Gamma \vdash (T,\ H,\ p ::= rv) \to (T',\ H''[a := (v,\ ts)],\ Skip)}\ \textsc{Assign}$$

Figure 1: The assignment rule

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* :: *gamma* ⇒ *config* ⇒ *bool* **where**
stuck Γ *cfg* ⟷ (*case cfg of*
(-, -, *Skip*) ⇒ *False* |
- ⇒ ¬(∃ *cfg*′. (Γ ⊢ *cfg* → *cfg*′)))

lemma *skip-final-noteq*[*simp*]: $\Gamma \vdash (T,\ H,\ c) \to (T,\ H',\ c') \Longrightarrow c \neq Skip$
⟨*proof*⟩

lemma *skip-final*[*simp*]: $\Gamma \vdash (T,\ H,\ Skip) \to (T',\ H',\ c) \Longrightarrow False$
⟨*proof*⟩

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

We show the transitivity of *com-sem-steps*.

lemma *com-seql-trans*:
$\Gamma \vdash (T,\ H,\ c1) \to^* (T',\ H',\ c1') \Longrightarrow \Gamma \vdash (T,\ H,\ c1;;\ c2) \to^* (T',\ H',\ c1';;\ c2)$
⟨*proof*⟩

4 Code examples

We need to assign a pre-allocated location to each variable to run a program. Currently we initialize those locations with zeros, but they will be uninitialized when we implement uninitialized slots.

abbreviation Γ_0 == λ-. *undefined*
fun *preallocate*′ :: *vname list* ⇒ *gamma* ∗ *tags* ∗ *heap* ⇒ *gamma* ∗ *tags* ∗ *heap*
where
preallocate′ [] *ret* = *ret* |
preallocate′ (*x* # *xs*) (Γ, *T*, *H*) =
(*let t* = *length T in*
let Γ′ = Γ(*x* := (*t*, *t*)) *in*
let T′ = *T* @ [*Unique*] *in*
let H′ = *H* @ [(*VInt 0*, [*t*])] *in*
preallocate′ *xs* (Γ′, *T*′, *H*′))
fun *preallocate* :: *vname list* ⇒ *gamma* ∗ *tags* ∗ *heap* **where**
preallocate xs = *preallocate*′ *xs* (Γ_0, [], [])

value *preallocate* [″*x*″, ″*y*″]
value *map* (*fst* (*preallocate* [″*x*″, ″*y*″])) [″*x*″, ″*y*″]

Let Isabelle run Unique programs by generating code for the semantics.

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

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

4.1 Invalidating a reference

Consider the following Rust program.

```
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 =
    (*Var* ''*root*'') ::= (*Use* ∘ *Constant* ∘ *VInt*) *42*;;
    (*Var* ''*px*'') ::= (*Reborrow* ∘ *Var*) ''*root*'';;
    ((*Deref* ∘ *Var*) ''*px*'') ::= (*Use* ∘ *Constant* ∘ *VInt*) *100*;;
    (*Var* ''*py*'') ::= (*Reborrow* ∘ *Var*) ''*root*'';;
    ((*Deref* ∘ *Var*) ''*py*'') ::= (*Use* ∘ *Constant* ∘ *VInt*) *200*

We can let Isabelle compute the program.

value *preallocate* [''*root*'', ''*px*'', ''*py*'']
abbreviation $\Gamma == \Gamma_0(''root'' := (0,\ 0),\ ''px'' := (1,\ 1),\ ''py'' := (2,\ 2))$
abbreviation *T* == [*Unique*, *Unique*, *Unique*]
abbreviation *H* == [(*VInt 0*, [*0*]), (*VInt 0*, [*1*]), (*VInt 0*, [*2*])]
lemma *preallocate* [''*root*'', ''*px*'', ''*py*''] = (Γ, *T*, *H*) ⟨*proof*⟩

values $\{(T',\ H')\ |$
    $T'\ H'\ c'.\ \Gamma \vdash (T,\ H,\ XY) \rightarrow^* (T',\ H',\ c')\}$

The following shows the tags and the heap at the end of execution.

abbreviation T_{XY} :: *tags* **where**
  T_{XY} == [*Unique*, *Unique*, *Unique*, *Unique*, *Unique*]
abbreviation H_{XY} :: *heap* **where**
  H_{XY} == [(*VInt 200*, [*0*, *4*]), (*Reference 0 3*, [*1*]), (*Reference 0 4*, [*2*])]

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*: $\Gamma \vdash (T,\ H,\ XY) \rightarrow^* (T_{XY},\ H_{XY},\ \mathit{Skip})$
  sorry

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

lemma *intermediate-xyx*: $\Gamma \vdash (T,\ H,\ XY;;\ ((\mathit{Deref} \circ \mathit{Var})\ ''px'') ::= (\mathit{Use} \circ \mathit{Constant} \circ \mathit{VInt})\ 300)$
  $\rightarrow^* (T_{XY},\ H_{XY},\ ((\mathit{Deref} \circ \mathit{Var})\ ''px'') ::= (\mathit{Use} \circ \mathit{Constant} \circ \mathit{VInt})\ 300)$
  ⟨*proof*⟩

lemma *stuck-xyx*: *stuck* Γ
   $(T_{XY},\ H_{XY},\ ((\mathit{Deref} \circ \mathit{Var})\ ''px'') ::= (\mathit{Use} \circ \mathit{Constant} \circ \mathit{VInt})\ 300)$
  ⟨*proof*⟩

lemma $\exists\, \mathit{cfg}.\ (\Gamma \vdash (T,\ H,\ XY;;\ ((\mathit{Deref} \circ \mathit{Var})\ ''px'') ::= (\mathit{Use} \circ \mathit{Constant} \circ \mathit{VInt})\ 300) \rightarrow^* \mathit{cfg})$
          $\wedge\ \mathit{stuck}\ \Gamma\ \mathit{cfg}$
  ⟨*proof*⟩

4.2 Swap

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

fun *swap* :: *place* ⇒ *place* ⇒ *com* **where**
  swap x y =
    $(\mathit{Var}\ ''reborrow\text{-}x'') ::= \mathit{Reborrow}\ x;;$
    $(\mathit{Var}\ ''reborrow\text{-}y'') ::= \mathit{Reborrow}\ y;;$
    $(\mathit{Var}\ ''tmp'') ::= (\mathit{Use} \circ \mathit{Place} \circ \mathit{Deref} \circ \mathit{Deref} \circ \mathit{Var})\ ''reborrow\text{-}x'';;$
     $((\mathit{Deref} \circ \mathit{Var})\ ''reborrow\text{-}x'') ::= (\mathit{Use} \circ \mathit{Place} \circ \mathit{Deref} \circ \mathit{Deref} \circ \mathit{Var})\ ''reborrow\text{-}y'';;$
    $((\mathit{Deref} \circ \mathit{Var})\ ''reborrow\text{-}y'') ::= (\mathit{Use} \circ \mathit{Place} \circ \mathit{Deref} \circ \mathit{Deref} \circ \mathit{Var})\ ''tmp''$

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

values $\{H \mid T\ H\ c.$
  $\Gamma_0(''reborrow\text{-}x'' := (0,\ 0),\ ''reborrow\text{-}y'' := (1,\ 1),\ ''tmp'' := (2,\ 2),\ ''a'' := (3,\ 3),\ ''b'' := (4,\ 4)) \vdash$
  $([\mathit{Unique},\ \mathit{Unique},\ \mathit{Unique}],$
   $[(\mathit{VInt}\ 0,\ [0]),\ (\mathit{VInt}\ 0,\ [1]),\ (\mathit{VInt}\ 0,\ [2]),\ (\mathit{VInt}\ 10000,\ [3]),\ (\mathit{VInt}\ 20000,\ [4])],$
   $\mathit{swap}\ (\mathit{Var}\ ''a'')\ (\mathit{Var}\ ''b'')) \rightarrow^* (T,\ H,\ c)\}$

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

end
theory *Rustv*

imports *Simpl.Vcg Simpl.Simpl-Heap*
begin

datatype *rust-error* = *invalid-ref*

type-synonym *tag* = *nat*
datatype *val* = *int-val int*
record *tagged-ref* =
pointer :: *ref*
tag :: *tag*

record *globals-ram* =
memory :: *val list*
tags :: *tag list list*
issued-tags :: *tag list*

fun *wf-tags* :: *tag list list* $\Rightarrow$ *bool* **where**
wf-tags [] = *True* |
wf-tags ([] # -) = *False* |
wf-tags (- # *t*) = *wf-tags t*
lemma *wf-tags-spec*: *wf-tags ts* $\longleftrightarrow$ ($\forall t \in$ *set ts*. $t \neq$ [])
$\langle proof \rangle$

lemma [*simp*, *intro*]: $[\![$*wf-tags ts*; $x \neq [\,]]\!] \Longrightarrow$ *wf-tags* (*ts* @ [*x*])
$\langle proof \rangle$

fun *collect-tags* :: *tag list list* $\Rightarrow$ *tag set* **where**
collect-tags ts = *foldr* (λ*ts accum*. *set ts* $\cup$ *accum*) *ts* {}
lemma *collect-tags-spec*: $t \in$ *collect-tags ts* $\longleftrightarrow$ ($\exists i <$ *length ts*. $t \in$ *set* (*ts* ! *i*))
$\langle proof \rangle$

declare *collect-tags.simps*[*simp del*]

lemma *collect-tags-update*[*simp*, *intro*]:
$t \in$ *collect-tags* (*ts*[*p* := *x*]) $\Longrightarrow t \in$ *collect-tags ts* $\vee\ t \in$ *set x*
$\langle proof \rangle$

lemma [*simp*, *intro*]: *collect-tags* (*ts* @ [*t*]) = *set t* $\cup$ (*collect-tags ts*)
$\langle proof \rangle$

lemma [*simp*, *intro*]: *finite* (*collect-tags ts*)
$\langle proof \rangle$

fun *wf-heap* :: *'a globals-ram-scheme* $\Rightarrow$ *bool* **where**
wf-heap s $\longleftrightarrow$
length (*memory s*) = *length* (*tags s*)
$\wedge$ *wf-tags* (*tags s*)
$\wedge$ *collect-tags* (*tags s*) $\subseteq$ *set* (*issued-tags s*)

fun *invalidate-children* :: *tagged-ref* ⇒ *'a globals-ram-scheme* ⇒ *'a globals-ram-scheme*
where
  invalidate-children r s =
    (*let p* = *Rep-ref* (*pointer r*);
        ts = *tags s* ! *p*;
        ts' = *dropWhile* ((≠) (*tag r*)) *ts in*
    s(| *tags* := (*tags s*)[*p* := *ts'*] |))

fun-cases *invalidate-childrenE*: *invalidate-children r s* = *s'*

lemma *dropWhile-hd-eq*[*simp*, *intro*]: *x* ∈ *set xs* ⟹ *hd* (*dropWhile* ((≠) *x*) *xs*) = *x*
⟨*proof*⟩

lemma *dropWhile-in*[*simp*, *intro*]: *x* ∈ *set xs* ⟹ *x* ∈ *set* (*dropWhile* ((≠) *x*) *xs*)
⟨*proof*⟩

lemma ⟦*invalidate-children r s* = *s'*; *Rep-ref* (*pointer r*) < *length* (*tags s*); *tag r* ∈ *set* ((*tags s'*) ! *Rep-ref* (*pointer r*))⟧
  ⟹ *hd* ((*tags s'*) ! *Rep-ref* (*pointer r*)) = *tag r*
  ⟨*proof*⟩

lemma *invalidate-children-memory*[*intro*]:
  invalidate-children r s = *s'* ⟹ *memory s'* = *memory s*
  ⟨*proof*⟩

fun *memwrite*
  :: *tagged-ref* ⇒ *val* ⇒ *'a globals-ram-scheme* ⇒ *'a globals-ram-scheme*
  where
  memwrite p v s =
    (*let memory* = *memory s in*
    let memory' = *memory*[*Rep-ref* (*pointer p*) := *v*] *in*
    s(| *memory* := *memory'* |))

lemma *memwrite-written*[*simp*]:
  fixes *p v s s'*
  assumes *s'* = *memwrite p v s*
        Rep-ref (*pointer p*) < *length* (*memory s*)
  shows (*memory s'*) ! *Rep-ref* (*pointer p*) = *v*
  ⟨*proof*⟩
lemma *memwrite-not-written*[*simp*]:
  fixes *p p' v s s'*
  assumes *s'* = *memwrite p v s*
        pointer p ≠ *pointer p'*
  shows (*memory s'*) ! *Rep-ref* (*pointer p'*) = (*memory s*) ! *Rep-ref* (*pointer p'*)
  ⟨*proof*⟩
lemma *memwrite-tags*:

fixes $p\ v\ s\ s'$
assumes $s' = memwrite\ p\ v\ s$
shows $tags\ s' = tags\ s$
⟨*proof*⟩

fun *new-tag* :: $'a$ *globals-ram-scheme* ⇒ *tag* **where**
new-tag s = *fold* (λt *accum*. *max* t *accum*) (*issued-tags* s) $0 + 1$

fun-cases *new-tag-elims*: *new-tag* $s = t$

lemma [*simp*, *intro*]: *new-tag* $s = t \Longrightarrow t \notin$ *set* (*issued-tags* s)
⟨*proof*⟩

lemma *fold-max-init*[*intro*]: *fold max xs* (n :: *nat*) $= m \Longrightarrow m \geq n$
⟨*proof*⟩

lemma *fold-max-elem*[*intro*]: *fold max xs* (n :: *nat*) $= m \Longrightarrow \forall x \in$ *set xs*. $m \geq x$
⟨*proof*⟩

lemma *max-fold-max*: $\forall x \in$ *set xs*. $m \geq x \Longrightarrow$ *fold max xs* $m = m$
⟨*proof*⟩

lemma
assumes
wf-heap s
new-tag $s = t$
$r <$ *length* (*tags* s)
shows $t \notin$ *set* (*tags* s ! r)
⟨*proof*⟩

fun *writable* :: *tagged-ref* ⇒ $'a$ *globals-ram-scheme* ⇒ *bool* **where**
writable $r\ s$ =
(*let* p = *Rep-ref* (*pointer* r) *in*
let t = *tag* r *in*
$p <$ *length* (*memory* s) $\wedge$ $t \in$ *set* (*tags* s ! p))

lemma *writable-update*[*simp*]: *writable* r (*memwrite* r' v s) = *writable* r s
⟨*proof*⟩

lemma *writable-invalidated*[*intro*]: *writable* $r\ s \Longrightarrow$ *writable* r (*invalidate-children* r s)
⟨*proof*⟩

fun *new-pointer* :: $'a$ *globals-ram-scheme* ⇒ *ref* **where**
new-pointer s = *Abs-ref* (*length* (*memory* s))

fun *heap-new* :: *val* ⇒ $'a$ *globals-ram-scheme* ⇒ *tagged-ref* $*$ $'a$ *globals-ram-scheme* **where**
heap-new $v\ s$ =

$(let\ p = new\text{-}pointer\ s;$
$t = new\text{-}tag\ s\ in$
$((\!|\ pointer = p,\ tag = t\ |\!),$
$s(\!|\ memory := memory\ s\ @\ [v],\ tags := tags\ s\ @\ [[t]],\ issued\text{-}tags := t\ \#\ issued\text{-}tags\ s\ |\!)))$

fun-cases *heap-new-elims*: $heap\text{-}new\ v\ s = (r,\ s')$

lemma *heap-new-writable*: $[\![wf\text{-}heap\ s;\ heap\text{-}new\ v\ s = (r',\ s')]\!] \Longrightarrow writable\ r'\ s'$
⟨*proof*⟩

lemma *heap-new-wf-heap-update*: $[\![wf\text{-}heap\ s;\ heap\text{-}new\ v\ s = (r',\ s')]\!] \Longrightarrow wf\text{-}heap\ s'$
⟨*proof*⟩

fun *reborrow* :: $tagged\text{-}ref \Rightarrow {}'a\ globals\text{-}ram\text{-}scheme \Rightarrow tagged\text{-}ref * {}'a\ globals\text{-}ram\text{-}scheme$
where
$reborrow\ r\ s =$
$(let\ p = Rep\text{-}ref\ (pointer\ r)\ in$
$let\ t = new\text{-}tag\ s\ in$
$let\ tags = (tags\ s)[p := t\ \#\ ((tags\ s)\ !\ p)]\ in$
$((\!|\ pointer = pointer\ r,\ tag = t\ |\!),\ s(\!|\ tags := tags,\ issued\text{-}tags := t\ \#\ issued\text{-}tags\ s\ |\!)))$

fun-cases *reborrow-elims*: $reborrow\ r\ s = (r',\ s')$

lemma *reborrow-pointer*: $reborrow\ r\ s = (r',\ s') \Longrightarrow pointer\ r' = pointer\ r$
⟨*proof*⟩

lemma *reborrow-update-heap*: $[\![wf\text{-}heap\ s;\ writable\ r\ s;\ reborrow\ r\ s = (r',\ s')]\!] \Longrightarrow wf\text{-}heap\ s'$
⟨*proof*⟩

lemma *reborrow-writable*: $[\![wf\text{-}heap\ s;\ writable\ r\ s;\ reborrow\ r\ s = (r',\ s')]\!] \Longrightarrow writable\ r\ s'$
⟨*proof*⟩

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.