Rustspec formalization

Denis Merigoux

September 20, 2020

1 Introduction

Rustspec is a subset of Rust designed for cryptographic specification. It is the successor of Hacspec [1]. Indeed, the strong typing environment of Rust and its low-level focus are more suitable to cryptographic specification than the subset of Python that was used for Hacspec. The main features of Rustspec are :

Pure functions By restricting borrowing and making full use of the `Copy` trait, Rustspec feels like functional programming.

Primitive array support Array manipulation is the cornerstone of cryptographic code, hence Rustspec arrays have primitive support in the language and a dedicated manipulation library.

Integers Rustspec supports all kinds of integers : word-sized integers (`u8`, `u32`, etc.), secret integers (constant-time by type abstraction design), natural integers (for field arithmetic).

Rustspec is concretely implemented as a Rust crate containing the types and functions of the language. However, Rustspec programs should also fall into a strict subset of Rust, that this document formalizes.

The Rustspec language, being embedded in Rust, shares with it some linearity properties of its type system. However, we severely restrict the borrowing feature: while retaining enough expressive power to write cryptographic specifications, the language behaves much simpler. The approach is conceptually similar to [2], although Rustspec is less expressive. Another inspiring attempt at formalizing a subset of Rust is [3].

The formalization presented here does not cover all the features of the Rustspec library. It focuses on the core of the languagestatement, its interesting type system and execution semantics. More precisely, we did not include the following elements :

- anything that can easily be desugared like assignment operators (`+=`, etc);
- the `if` expression, that coexists with the `if` statement;
- all the various types of integers, the casts between them and all the operators to manipulate them (we only have basic `int` type with arithmetic);
- type inference
- polymorphism and traits, as we assume that the Rust compiler does the job of monomorphizing everything for us.

2 Syntax

The Rustspec language is a subset of Rust. Apart from the basic `int` and `bool` types, the language operates on arrays of fixed size, known at compilation (`array!`) or not (`Seq`). The biggest limitation compared to the full Rust language is the borrowing patterns that we allow. Indeed, we restrict borrowing to only immutable borrowing (`&`), and we also restrict where this borrowing can happen: only in function arguments or at function call sites. Additionally, functions cannot return types containing references. Because of these restrictions, there is not much you can do with types subject to linearity in Rustspec, compared to what you can do in Rust. However, we argue that it is enough for cryptographic specifications.

Program	p	::=	$[i]^*$	list of items
Item	i	::=	`array!(` t`,` μ`,` $n \in \mathbb{N}$ `);`	array type declaration
		|	`fn` f`(` $[d]^+$ `) ->` μ b	function declaration
Argument declaration	d	::=	x `:` τ	
Reference-free type	μ	::=	`unit` | `bool` | `int`	base types
		|	`Seq<` μ `>`	sequence
		|	t	type variable
		|	`(` $[\mu]^+$ `)`	tuple
Type	τ	::=	μ	base types
		|	`&`μ	immutable reference
Block	b	::=	`{` $[s;]^+$ `}`	
Statement	s	::=	`let` x `:` τ `=` e	let binding
		|	x `=` e	variable reassignment
		|	`if` e `then` b (`else` b)	conditional statements
		|	`for` x `in` e `..` e b	for loop (integers only)
		|	x`[` e `]` `=` e	array update
		|	e	return expression
		|	b	statement block
Expression	e	::=	`()` | `true` | `false`	unit and boolean literals
		|	$n \in \mathbb{N}$	integer literal
		|	x	variable
		|	f`(` $[a]^+$ `)`	function call
		|	$e \odot e$	binary operations
		|	$\oslash$ e	unary operations
		|	`(` $[e]^+$ `)`	tuple constructor
		|	e`.`$(n \in \mathbb{N})$	tuple field access
		|	e`[` e `]`	array or seq index
Function argument	a	::=	e	linear argument
		|	`&`e	call-site borrowing
Binary operators	$\odot$	::=	`+` | `-` | `*` | `/` | `&&` | `||`	
Unary operators	$\oslash$	::=	`-` | `~`	

3 Operational semantics

The operational semantics for this language are standard, using big-step evaluation. Indeed, the order of evaluation of tuple elements or function arguments does not matter, since expressions

of the language are pure. The borrowing does not have any effect on the operational semantics, but it will affect the typing judgment later.

Please note that the contexts Ω is considered as an unordered set rather than an ordered list. As such, the rules have the relevant elements appear at the end or the beginning of the context without loss of generality.

Value	v	$::=$	`true` | `false`	boolean
		$\mid$	$n \in \mathbb{N}$	integer
		$\mid$	$[\ [v]^*\ ]$	array or sequence
		$\mid$	$(\ [v]^*\ )$	tuple
Evaluation context	Ω	$::=$	$\varnothing$	empty context
		$\mid$	$x \mapsto v, \Omega$	variable value

Expression evaluation	$p; \Omega \vdash e \Downarrow v$
Function argument evaluation	$p; \Omega \vdash a \Downarrow v$
Statement evaluation	$p; \Omega \vdash s \Downarrow v \Rrightarrow \Omega$
Block evaluation	$p; \Omega \vdash b \Downarrow v \Rrightarrow \Omega$

EvalUnit

$$\frac{}{p; \Omega \vdash () \Downarrow ()}$$

EvalBool

$$\frac{b \in \{\texttt{true}, \texttt{false}\}}{p; \Omega \vdash b \Downarrow b}$$

EvalInt

$$\frac{n \in \mathbb{N}}{p; \Omega \vdash n \Downarrow n}$$

EvalVar

$$\frac{}{p; x \mapsto v, \Omega \vdash x \Downarrow v}$$

EvalFuncArg

$$\frac{p; \Omega \vdash e \Downarrow v}{p; \Omega \vdash \&e \Downarrow v}$$

EvalFuncCall

$$\frac{\texttt{fn}\ f(\ x_1 : \tau_1, \ldots, x_n : \tau_n\ )\ \texttt{->}\ \mu\ b \in p \qquad \forall i \in [\![1, n]\!],\ p; \Omega \vdash a_i \Downarrow v_i \qquad p; x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \vdash b \Downarrow v}{p; \Omega \vdash f(\ a_1, \ldots, a_n\ ) \Downarrow v}$$

EvalBinaryOp

$$\frac{p; \Omega \vdash e_1 \Downarrow v_1 \qquad p; \Omega \vdash e_2 \Downarrow v_2}{p; \Omega \vdash e_1 \odot e_2 \Downarrow v_1 \odot v_2}$$

EvalUnaryOp

$$\frac{p; \Omega \vdash e \Downarrow v}{p; \Omega \vdash \oslash e \Downarrow \oslash v}$$

EvalTuple

$$\frac{\forall i \in [\![1, n]\!],\ p; \Omega \vdash e_i \Downarrow v_i}{p; \Omega \vdash (\ e_1, \ldots, e_n\ ) \Downarrow (\ v_1, \ldots, v_n\ )}$$

EvalTupleAccess

$$\frac{p; \Omega \vdash e \Downarrow (\ v_1, \ldots, v_m\ ) \qquad n \in [\![1, m]\!]}{p; \Omega \vdash e.n \Downarrow v_n}$$

EvalArrayAccess

$$\frac{p; \Omega \vdash e_1 \Downarrow [\ v_0, \ldots, v_m\ ] \qquad p; \Omega \vdash e_2 \Downarrow n \qquad n \in [\![0, m]\!]}{p; \Omega \vdash e_1[\ e_2\ ] \Downarrow v_n \Rrightarrow \Omega}$$

EvalLet

$$\frac{p; \Omega \vdash e \Downarrow v}{p; \Omega \vdash \texttt{let}\ x : \tau = e \Downarrow () \Rrightarrow x \mapsto v, \Omega}$$

EvalReassign

$$\frac{p; x \mapsto v, \Omega \vdash e \Downarrow v'}{p; x \mapsto v, \Omega \vdash x = e \Downarrow () \Rrightarrow x \mapsto v', \Omega}$$

EvalIfThenTrue

$$\frac{p; \Omega \vdash e_1 \Downarrow \texttt{true} \qquad p; \Omega \vdash b \Downarrow () \Rrightarrow \Gamma'}{p; \Omega \vdash \texttt{if}\ e_1\ b \Downarrow () \Rrightarrow \Gamma'}$$

EvalIfThenFalse

$$\frac{p; \Omega \vdash e_1 \Downarrow \texttt{false}}{p; \Omega \vdash \texttt{if}\ e_1\ b \Downarrow () \Rrightarrow \Gamma}$$

EvalIfThenElseTrue

$$\frac{p; \Omega \vdash e_1 \Downarrow \texttt{true} \qquad p; \Omega \vdash b \Downarrow () \Rrightarrow \Gamma'}{p; \Omega \vdash \texttt{if}\ e_1\ b\ \texttt{else}\ b' \Downarrow () \Rrightarrow \Gamma'}$$

EvalIfThenElseFalse

$$\frac{p; \Omega \vdash e_1 \Downarrow \texttt{false} \qquad p; \Omega \vdash b' \Downarrow () \Rrightarrow \Gamma'}{p; \Omega \vdash \texttt{if}\ e_1\ b\ \texttt{else}\ b' \Downarrow () \Rrightarrow \Gamma'}$$

EvalForLoop

$$\frac{p;\,\Omega \vdash e_1 \Downarrow n \quad p;\,\Omega \vdash e_2 \Downarrow m \qquad \Omega_n = \Omega \quad \forall i \in [\![n, m-1]\!],\ p;\,\Omega_i \vdash b \Downarrow () \Rrightarrow \Omega_{i+1}}{p;\,\Omega \vdash \texttt{for}\ x\ \texttt{in}\ e_1\ ..\ e_2\ b \Downarrow () \Rrightarrow \Omega_m}$$

EvalArrayUpd

$$\frac{p;\,x \mapsto [\,v_0, \ldots, v_n\,], \Omega \vdash e_1 \Downarrow m \qquad m \in [\![0, n]\!] \qquad p;\,x \mapsto [\,v_0, \ldots, v_n\,], \Omega \vdash e_2 \Downarrow v}{p;\,x \mapsto [\,v_0, \ldots, v_n\,], \Omega \vdash x\,[\,e_1\,] = e_2 \Downarrow () \Rrightarrow x \mapsto [\,v_0, \ldots, v_{m-1}, v, v_{m+1}, \ldots, v_n\,], \Omega}$$

EvalExprStmt

$$\frac{p;\,\Omega \vdash e \Downarrow v}{p;\,\Omega \vdash e \Downarrow v \Rrightarrow \Omega}$$

EvalBlock

$$\frac{p;\,\Omega \vdash s_1 \Downarrow () \Rrightarrow \Omega' \qquad p;\,\Omega' \vdash \{\,s_2\,;\,\ldots\,;\,s_n\,\} \Downarrow v \Rrightarrow \Omega''}{p;\,\Omega \vdash \{\,s_1\,;\,\ldots\,;\,s_n\,\} \Downarrow v \Rrightarrow \Omega''}$$

EvalBlockOne

$$\frac{p;\,\Omega \vdash s \Downarrow v \Rrightarrow \Omega'}{p;\,\Omega \vdash \{\,s\,\} \Downarrow v \Rrightarrow \Omega'}$$

EvalBlockAsStatement

$$\frac{p;\,\Omega \vdash b \Downarrow v \Rrightarrow \Omega'}{p;\,\Omega \vdash b \Downarrow v \Rrightarrow \Omega'}$$

Function evaluation $p \vdash f\,(\,v_1, \ldots, v_n\,) \Downarrow v$

EvalFunc

$$\frac{\texttt{fn}\ f\,(\,x_1 : \tau_1, \ldots, x_n : \tau_n\,)\ \texttt{->}\ \mu\ b \in p \qquad p;\,x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \vdash b \Downarrow v}{p \vdash f\,(\,v_1, \ldots, v_n\,) \Downarrow v}$$

4 Typing

Contrasting with the operational semantics, the typing rules are fairly complex and restrictive. The reason for this complexity is our objective to stick to Rust behavior.

The typing environnement of Rustspec is fairly standard. We need a type dictionary to enforce the named type discipline of Rust that covers the types declared by the `array!` macro. Please note that the contexts Γ and Δ are considered as unordered sets rather than ordered lists. As such, the rules have the relevant elements appear at the end or the beginning of those contexts without loss of generality.

Typing context	Γ	::=	$\varnothing$	empty context
		|	$x : \tau, \Gamma$	variable
		|	$f : (\,[\tau]^+\,) \to \mu, \Gamma$	function
Type dictionary	Δ	::=	$\varnothing$	empty dictionnary
		|	$t \to [\,\mu; n \in \mathbb{N}\,], \Delta$	array type

The restrictions on borrowing lead to severe limitations on how we can manipulate values of linear type in our language, rendering it quite useless at first sight. Indeed, when you receive a reference as a function argument, you can only use it in expressions and perform identity let bindings with it. You cannot store it in memory or in a tuple and pass it around indirectly in your program. This behavior is well-suited for input and output buffers in cryptographic code.

However, in the spirit of Rust, we introduce an escape hatch from linearity under the form of the `Copy` trait implementation. This trait, that is primitive to the Rust language, is used to distinguish the values that are "cheap" to copy. This concerns all the reference-free μ types except `Seq`, whose size is not known at compilation time (and thus can be arbitrarily large). Paradoxically, `array!` types that can be as large as their `Seq` counterparts do benefit from

the `Copy` trait; we replicate here the behavior of Rust. Indeed, because the lenght is known at compilation time, the code generation backend of Rust (LLVM) can optimize the representation of the array in memory, especially if the size is small.

With this setup, both `array!` and `Seq` represent a table of data. The moral difference between them is that `array!` is a table passed by value, whereas `Seq` is a tabled passed by reference (immutable). In the cryptographic specifications, `Seq` is mostly used for input and output messages, whose length is known only at runtime. Fixed-size chunks or blocks of data are rather implemented using `array!`.

Implementing the `Copy` trait $\quad \Delta \vdash \tau : \texttt{Copy}$

CopyUnit
$$\frac{}{\Delta \vdash \texttt{unit} : \texttt{Copy}}$$

CopyBool
$$\frac{}{\Delta \vdash \texttt{bool} : \texttt{Copy}}$$

CopyInt
$$\frac{}{\Delta \vdash \texttt{int} : \texttt{Copy}}$$

CopyTuple
$$\frac{\Delta \vdash \tau_1 : \texttt{Copy} \quad \cdots \quad \Delta \vdash \tau_n : \texttt{Copy}}{\Delta \vdash (\ \tau_1, \ldots, \tau_n\ ) : \texttt{Copy}}$$

CopyArray
$$\frac{\Delta \vdash \mu : \texttt{Copy}}{t \to [\ \mu; n\ ], \Delta \vdash t : \texttt{Copy}}$$

Because Rust has an affine type system, Rustspec also enjoys an affine typing context with associated splitting rules (SplitLinear). Please note that immutable references values can be duplicated freely in the context (SplitDuplicable). During an elaboration phase inside the Rust compiler, the linearity of the type system gets circumvented for `Copy` values with the insertion of `clone()` functions call that perform a copy of the value wherever the linear type system forces a copy of the value to be made. We formalize this behavior here by allowing `Copy` types duplication in the typing context, like immutable references (SplitCopy). Lastly, functions are always duplicable in the context (SplitFunction).

Context splitting $\quad \Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2$

SplitEmpty
$$\frac{}{\Delta \vdash \varnothing = \varnothing \circ \varnothing}$$

SplitLinear1
$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2}{\Delta \vdash x : \tau, \Gamma = (\ x : \tau, \Gamma_1\ ) \circ \Gamma_2}$$

SplitLinear2
$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2}{\Delta \vdash x : \tau, \Gamma = \Gamma_1 \circ (\ x : \tau, \Gamma_2\ )}$$

SplitDuplicable
$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2}{\Delta \vdash x : \&\tau, \Gamma = (\ x : \&\tau, \Gamma_1\ ) \circ (\ x : \&\tau, \Gamma_2\ )}$$

SplitCopy
$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2 \qquad \Delta \vdash \tau : \texttt{Copy}}{\Delta \vdash x : \tau, \Gamma = (\ x : \tau, \Gamma_1\ ) \circ (\ x : \tau, \Gamma_2\ )}$$

SplitFunction
$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \Gamma_2}{\Delta \vdash f : (\ \mu_1, \ldots, \mu_n\ ) \to \tau, \Gamma = (\ f : (\ \mu_1, \ldots, \mu_n\ ) \to \tau, \Gamma_1\ ) \circ (\ f : (\ \mu_1, \ldots, \mu_n\ ) \to \tau, \Gamma_2\ )}$$

We can now proceed to the main typing judgments. TypVarLinear and TypVarDup reflect the variable typing present in the context. TypTupleCons only allows non-reference values inside a tuple, with a linear context splitting to check each term of the tuple. TypArrayAccess,

TypSeqAccess and TypSeqRefAccess specify the array indexing syntax, which is overloaded to work with both `array!`, `Seq` and `&Seq`. This corresponds to the implementing of the `Index` trait in Rust.

The function call rule, TypFuncCall, is the most complex rule of the typing judgment, because it contains the restricted borrowing form allowed in Rustspec. First, note that the context is split for typechecking the arguments of the function, because a linear value cannot be used in two arguments. Next, TypFunArgLin ensures that the arguments are well-typed. However, if an argument is borrowed at call-site, then TypFunArgDup checks the value that is being borrowed under the reference. In our degenerate pattern of borrowing, TypFunArgDup is doing the work of the Rust borrow checker. The last rules for binary and unary operations are standard.

Value typing	$\Gamma; \Delta \vdash v : \tau$
Expression typing	$\Gamma; \Delta \vdash e : \tau$
Function argument typing	$\Gamma; \Delta \vdash a \sim \tau$

TypUnit
$$\frac{}{\Gamma; \Delta \vdash () : \texttt{unit}}$$

TypBool
$$\frac{b \in \{\texttt{true}, \texttt{false}\}}{\Gamma; \Delta \vdash b : \texttt{bool}}$$

TypInt
$$\frac{n \in \mathbb{N}}{\Gamma; \Delta \vdash n : \texttt{int}}$$

TypSeqValue
$$\frac{\forall i \in [\![1, n]\!],\ \Gamma; \Delta \vdash v_i : \mu}{\Gamma; \Delta \vdash [\, v_1, \ldots, v_n \,] : \texttt{Seq}\langle\, \mu \,\rangle}$$

TypArrayValue
$$\frac{\forall i \in [\![1, n]\!],\ \Gamma; \Delta \vdash v_i : \mu}{\Gamma; \Delta \vdash [\, v_1, \ldots, v_n \,] : [\, \mu; n \,]}$$

TypVarLinear
$$\frac{}{x : \tau, \Gamma; \Delta \vdash x : \tau}$$

TypVarDup
$$\frac{}{x : \&\tau, \Gamma; \Delta \vdash x : \&\tau}$$

TypTupleCons
$$\frac{\Delta \vdash \Gamma = \Gamma_1 \circ \cdots \circ \Gamma_n \qquad \forall i \in [\![1, n]\!],\ \Gamma; \Delta \vdash e_i : \mu_i}{\Gamma; \Delta \vdash (\, e_1, \ldots, e_n \,) : (\, \mu_1, \ldots, \mu_n \,)}$$

TypTupleAccess
$$\frac{\Gamma; \Delta \vdash e : (\, \mu_1, \ldots, \mu_m \,) \qquad n \in [\![1, m]\!]}{\Gamma; \Delta \vdash e.n : \mu_n}$$

TypArrayAccess
$$\frac{\Gamma; t \to [\, \mu; n \,], \Delta \vdash e_1 : t \qquad \Gamma; t \to [\, \mu; n \,], \Delta \vdash e_2 : \texttt{int}}{\Gamma; t \to [\, \mu; n \,], \Delta \vdash e_1 [\, e_2 \,] : \mu}$$

TypSeqAccess
$$\frac{\Gamma; \Delta \vdash e_1 : \texttt{Seq}\langle\, \mu \,\rangle \qquad \Gamma; \Delta \vdash e_2 : \texttt{int}}{\Gamma; \Delta \vdash e_1 [\, e_2 \,] : \mu}$$

TypSeqRefAccess
$$\frac{\Gamma; \Delta \vdash e_1 : \&\texttt{Seq}\langle\, \mu \,\rangle \qquad \Gamma; \Delta \vdash e_2 : \texttt{int}}{\Gamma; \Delta \vdash e_1 [\, e_2 \,] : \mu}$$

TypFuncCall
$$\frac{\Gamma = f : (\, \mu_1, \ldots, \mu_n \,) \to \tau, \Gamma' \qquad \Delta \vdash \Gamma = \Gamma_1 \circ \cdots \circ \Gamma_n \qquad \forall i \in [\![1, n]\!],\ \Gamma_i; \Delta \vdash a_i \sim \mu_i}{\Gamma; \Delta \vdash f (\, a_1, \ldots, a_n \,) : \tau}$$

TypFunArgLin
$$\frac{\Gamma; \Delta \vdash e : \tau}{\Gamma; \Delta \vdash e \sim \tau}$$

TypFunArgDup
$$\frac{\Gamma; \Delta \vdash e : \mu}{\Gamma; \Delta \vdash \&e \sim \&\mu}$$

TypBinopInt
$$\frac{\Gamma; \Delta \vdash e_1 : \texttt{int} \qquad \Gamma; \Delta \vdash e_2 : \texttt{int} \qquad \odot \in \{\, +, -, *, / \,\}}{\Gamma; \Delta \vdash e_1 \odot e_2 : \texttt{int}}$$

$$\textsc{TypBinopBool}\ \frac{\Gamma;\Delta \vdash e_1 : \texttt{bool} \qquad \Gamma;\Delta \vdash e_2 : \texttt{bool} \qquad \odot \in \{\ \texttt{\&\&}\ ,\ \texttt{||}\ \}}{\Gamma;\Delta \vdash e_1\ \odot\ e_2 : \texttt{bool}} \qquad \textsc{TypUnopInt}\ \frac{\Gamma;\Delta \vdash e : \texttt{int} \qquad \oslash \in \{\ \texttt{-}\ \}}{\Gamma;\Delta \vdash \oslash\ e : \texttt{int}}$$

$$\textsc{TypUnopBool}\ \frac{\Gamma;\Delta \vdash e : \texttt{bool} \qquad \oslash \in \{\ \texttt{\~{}}\ \}}{\Gamma;\Delta \vdash \oslash\ e : \texttt{bool}}$$

Let's now move to the statement typing. We've chosen statements here rather than nested expressions because of the Rust behavior of the `if` statement and the `for` loop. A list of statement corresponds to a block, introduced in Rust by $\{\cdots\}$. The single statement typing judgment produces a new Γ' because of variable definitions inside a block. Single statements also yield back a type, because the last statement of the function is also the return value of the function. All statement type except the last one should be `unit`.

The rule TypLet introduces a new mutable local variable, that can later be reassigned (TypReassign) in the program. In Rust, the `mut` indicates that the local variable is mutable, in its absence, variable reassignments are prohibited. In this formalization, all variables are mutable for simplification. The main use of mutable local variables is for variables that are mutated inside a `for` loop. Indeed, because `for` loops are restricted to integer range iteration, we cannot express what would normally be a fold without these mutable variables. Because the mutable variables are local to a block, we do not need to formalize a full-fledged heap for the operational semantics. Rather, we will model them as a limited piece of state that gets passed around during execution.

Next, TypArrayAssign and TypSeqAssign define the overloading of the array update syntax that works for both `array!` and `Seq`. Note that TypIfThenElse use the same context Γ for the two branches of the conditional.

Statement typing	$\Gamma;\Delta \vdash s : \tau \Rrightarrow \Gamma'$
Block typing	$\Gamma;\Delta \vdash b : \tau$

$$\textsc{TypLet}\ \frac{\Gamma;\Delta \vdash e : \tau \qquad x \notin \Gamma}{\Gamma;\Delta \vdash \texttt{let}\ x\ :\ \tau = e : \texttt{unit} \Rrightarrow \Gamma, x : \tau}$$

$$\textsc{TypReassign}\ \frac{\Gamma, x : \tau;\Delta \vdash e : \tau}{\Gamma, x : \tau;\Delta \vdash x = e : \texttt{unit} \Rrightarrow \Gamma, x : \tau}$$

$$\textsc{TypArrayAssign}\ \frac{x : t, \Gamma; t \to [\ \mu; n\ ], \Delta \vdash e_1 : \texttt{int} \qquad x : t, \Gamma; t \to [\ \mu; n\ ], \Delta \vdash e_2 : \mu}{x : t, \Gamma; t \to [\ \mu; n\ ], \Delta \vdash x[\ e_1\ ] = e_2 : \texttt{unit} \Rrightarrow \Gamma}$$

$$\textsc{TypSeqAssign}\ \frac{x : \texttt{Seq}\!<\mu>, \Gamma;\Delta \vdash e_1 : \texttt{int} \qquad x : \texttt{Seq}\!<\mu>, \Gamma;\Delta \vdash e_2 : \mu}{x : \texttt{Seq}\!<\mu>, \Gamma;\Delta \vdash x[\ e_1\ ] = e_2 : \texttt{unit} \Rrightarrow \Gamma}$$

$$\textsc{TypIfThen}\ \frac{\Gamma;\Delta \vdash e\ :\ \texttt{bool} \qquad \Gamma;\Delta \vdash b : \texttt{unit}}{\Gamma;\Delta \vdash \texttt{if}\ e\ \texttt{then}\ b : \texttt{unit} \Rrightarrow \Gamma}$$

$$\textsc{TypIfThenElse}\ \frac{\Gamma;\Delta \vdash e : \texttt{bool} \qquad \Gamma;\Delta \vdash b : \texttt{unit} \qquad \Gamma;\Delta \vdash b' : \texttt{unit}}{\Gamma;\Delta \vdash \texttt{if}\ e\ \texttt{then}\ b\ \texttt{else}\ b' : \texttt{unit} \Rrightarrow \Gamma}$$

$$\textsc{TypForLoop}\ \frac{\Gamma;\Delta \vdash e_1 : \texttt{int} \qquad \Gamma;\Delta \vdash e_2 : \texttt{int} \qquad \Gamma, x : \texttt{int};\Delta \vdash b : \texttt{unit}}{\Gamma;\Delta \vdash \texttt{for}\ x\ \texttt{in}\ e_1\ ..\ e_2\ b : \texttt{unit} \Rrightarrow \Gamma}$$

$$\textsc{TypExpToStmt}\ \frac{\Gamma;\Delta \vdash e : \tau}{\Gamma;\Delta \vdash e : \tau \Rrightarrow \Gamma}$$

$$\textsc{TypBlock}\ \frac{\Gamma;\Delta \vdash s_1 : \texttt{unit} \Rrightarrow \Gamma' \qquad \Gamma';\Delta \vdash \{\ s_2 ; \ldots ;\ s_n\ \} : \tau}{\Gamma;\Delta \vdash \{\ s_1 ; \ldots ;\ s_n\ \} : \tau}$$

$$\textsc{TypBlockOne}\ \frac{\Gamma;\Delta \vdash s_1 : \tau \Rrightarrow \Gamma'}{\Gamma;\Delta \vdash \{\ s_1\ \} : \tau}$$

$$\textsc{TypBlockAsStatement}\ \frac{\Gamma;\Delta \vdash b : \tau}{\Gamma;\Delta \vdash b : \tau \Rrightarrow \Gamma}$$

A Rustspec program is a list of items i. Their typing judgment produces both a new Γ and Δ, because an item introduces either a new function or a new named type. Please note, as mentionned before, that the return type of functions is restricted to μ, as returning a reference is forbidden. The TypFnDecl also means that recursion is forbidden in Rustspec, since f is not passed in its typing context.

$$\text{Item typing} \quad \Gamma;\Delta \vdash i \Rrightarrow \Gamma';\Delta'$$

$$\textsc{TypArrayDecl}\ \frac{}{\Gamma;\Delta \vdash \texttt{array!}(\ t,\ \mu,\ n\ ) \Rrightarrow \Gamma;\Delta, t \to [\ \mu; n\ ]}$$

$$\textsc{TypFnDecl}\ \frac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n;\Delta \vdash b : \mu}{\Gamma;\Delta \vdash \texttt{fn}\ f(\ x_1 : \tau_1, \ldots, x_n : \tau_n\ )\ \texttt{->}\ \mu\ b \Rrightarrow \Gamma, f : (\ \tau_1, \ldots, \tau_n\ ) \to \mu;\Delta}$$

References

[1] Karthikeyan Bhargavan, Franziskus Kiefer, and Pierre-Yves Strub. hacspec: Towards verifiable crypto standards. In Cas Cremers and Anja Lehmann, editors, *Security Standardisation Research*, pages 1–20, Cham, 2018. Springer International Publishing.

[2] Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. Kindly bent to free us, 2019.

[3] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of rust. *CoRR*, abs/1903.00982, 2019.