Rust

A safe, concurrent, practical language.

Graydon Hoare
Mozilla Foundation

Table of Contents

1 Disclaimer

To the reader,

Rust is a work in progress. The language continues to evolve as the design shifts and is fleshed out in working code. Certain parts work, certain parts do not, certain parts will be removed or changed.

This manual is a snapshot written in the present tense. Some features described do not yet exist in working code. Some may be temporary. It is a *draft*, and we ask that you not take anything you read here as either definitive or final. The manual is to help you get a sense of the language and its organization, not to serve as a complete specification. At least not yet.

If you have suggestions to make, please try to focus them on *reductions* to the language: possible features that can be combined or omitted. At this point, every "additive" feature we're likely to support is already on the table. The task ahead involves combining, trimming, and implementing.

2 Introduction

> We have to fight chaos, and the most effective way of doing that is to prevent its emergence.
>
> - Edsger Dijkstra

Rust is a curly-brace, block-structured statement language. It visually resembles the C language family, but differs significantly in syntactic and semantic details. Its design is oriented toward concerns of "programming in the large", that is, of creating and maintaining *boundaries* – both abstract and operational – that preserve large-system *integrity*, *availability* and *concurrency*.

It supports a mixture of imperative procedural, concurrent actor, object oriented and pure functional styles. Rust also supports generic programming and metaprogramming, in both static and dynamic styles.

2.1 Goals

The language design pursues the following goals:

- Compile-time error detection and prevention.
- Run-time fault tolerance and containment.
- System building, analysis and maintenance affordances.
- Clarity and precision of expression.
- Implementation simplicity.
- Run-time efficiency.
- High concurrency.

Note that most of these goals are *engineering* goals, not showcases for sophisticated language technology. Most of the technology in Rust is *old* and has been seen decades earlier in other languages.

All new languages are developed in a technological context. Rust's goals arise from the context of writing large programs that interact with the internet – both servers and clients – and are thus much more concerned with *safety* and *concurrency* than older generations of program. Our experience is that these two forces do not conflict; rather they drive system design decisions toward extensive use of *partitioning* and *statelessness*. Rust aims to make these a more natural part of writing programs, within the niche of lower-level, practical, resource-conscious languages.

2.2 Sales Pitch

The following comprises a brief "sales pitch" overview of the salient features of Rust, relative to other languages.

- No `null` pointers

  The initialization state of every slot is statically computed as part of the typestate system (see below), and requires that all slots are initialized before use. There is no `null` value; uninitialized slots are uninitialized, and can only be written to, not read.

  The common use for `null` in other languages – as a sentinel value – is subsumed into the more general facility of disjoint union types. A program must explicitly model its use of such types.

- Lightweight tasks with no shared mutable state

  Like many *actor* languages, Rust provides an isolation (and concurrency) model based on lightweight tasks scheduled by the language runtime. These tasks are very inexpensive and statically unable to mutate one another's local memory. Breaking the rule of task isolation is only possible by calling external (C/C++) code.

  Inter-task communication is typed, asynchronous and simplex, based on passing messages over channels to ports. Transmission can be rate-limited or rate-unlimited. Selection between multiple senders is pseudo-randomized on the receiver side.

- Predictable native code, simple runtime

  The meaning and cost of every operation within a Rust program is intended to be easy to model for the reader. The code should not "surprise" the programmer once it has been compiled.

  Rust compiles to native code. Rust compilation units are large and the compilation model is designed around multi-file, whole-library or whole-program optimization. The compiled units are standard loadable objects (ELF, PE, Mach-O) containing standard metadata (DWARF) and are compatible with existing, standard low-level tools (disassemblers, debuggers, profilers, dynamic loaders).

  The Rust runtime library is a small collection of support code for scheduling, memory management, inter-task communication, reflection and runtime linkage. This library is written in standard C++ and is quite straightforward. It presents a simple interface to embeddings. No research-level virtual machine, JIT or garbage collection technology is required. It should be relatively easy to adapt a Rust front-end on to many existing native toolchains.

- Integrated system-construction facility

  The units of compilation of Rust are multi-file amalgamations called *crates*. A crate is described by a separate, declarative type of source file that guides the compilation of the crate, its packaging, its versioning, and its external dependencies. Crates are also the units of distribution and loading. Significantly: the dependency graph of crates is *acyclic* and *anonymous*: there is no global namespace for crates, and module-level recursion cannot cross crate barriers.

Unlike many languages, individual modules do *not* carry all the mechanisms or restrictions of crates. Modules and crates serve different roles.

- Stack-based iterators

  Rust provides a type of function-like multiple-invocation iterator that is very efficient: the iterator state lives only on the stack and is tightly coupled to the loop that invoked it.

- Direct interface to C code

  Rust can load and call many C library functions simply by declaring them. Calling a C function statically marks a function as "unsafe", unless the task calling the unsafe function is further isolated within an external "heavyweight" operating-system subprocess. Every "unsafe" function or module in a Rust compilation unit must be explicitly authorized in the crate file.

- Structural algebraic data types

  The Rust type system is structural rather than nominal, and contains the standard assortment of useful "algebraic" type constructors from functional languages, such as function types, tuples, record types, vectors, and tagged disjoint unions. Structural types may be *pattern-matched* in an `alt` statement.

- Generic code

  Rust supports a simple form of parametric polymorphism: functions, iterators, types and objects can be parametrized by other types.

- Argument binding

  Rust provides a mechanism of partially binding arguments to functions, producing new functions that accept the remaining un-bound arguments. This mechanism combines some of the features of lexical closures with some of the features of currying, in a smaller and simpler package.

- Local type inference

  To save some quantity of programmer key-pressing, Rust supports local type inference: signatures of functions, objects and iterators always require type annotation, but within the body of a function or iterator many slots can be declared `auto` and Rust will infer the slot's type from its uses.

- Structural object system

  Rust has a lightweight object system based on structural object types: there is no "class hierarchy" nor any concept of inheritance. Method overriding and object restriction are performed explicitly on object values, which are little more than order-insensitive records of methods sharing a common private value. Objects can be mutable or immutable, and immutable objects can have destructors.

- Dynamic type

Rust includes support for slots of a top type, `any`, that can hold any type of value whatsoever. An `any` slot is a pair of a type code and an exterior value of that type. Injection into an `any` and projection by type-case-selection is integrated into the language.

- Dynamic metaprogramming (reflection)

  Rust supports run-time reflection on the structure of a crate, using a combination of custom descriptor structures and the DWARF metadata tables used to support crate linkage and other runtime services.

- Static metaprogramming (syntactic extension)

  Rust supports a system for syntactic extensions that can be loaded into the compiler, to implement user-defined notations, macros, program-generators and the like. These notations are *marked* using a special form of bracketing, such that a reader unfamiliar with the extension can still parse the surrounding text by skipping over the bracketed "extension text".

- Idempotent failure

  If a task fails due to a signal, or if it executes the special `fail` statement, it enters the *failing* state. A failing task unwinds its control stack, frees all of its owned resources (executing destructors) and enters the *dead* state. Failure is idempotent and non-recoverable.

- Signal handling

  Rust has a system for propagating task-failures and other spontaneous events between tasks. Some signals can be trapped and redirected to channels; other signals are fatal and result in task-failure. Tasks can designate other tasks to handle signals for them. This permits organizing tasks into mutually-supervising or mutually-failing groups.

- Deterministic destruction

  Immutable objects can have destructor functions, which are executed deterministically in top-down ownership order, as control frames are exited and/or objects are otherwise freed from data structures holding them. The same destructors are run in the same order whether the object is deleted by unwinding during failure or normal execution.

  Similarly, the rules for freeing immutable memory are deterministic and predictable: on scope-exit or structure-release, interior slots are released immediately, exterior slots have their reference count decreased and are released if the count drops to zero. Alias slots are not affected by scope exit.

  Mutable memory is local to a task, and is subject to per-task garbage collection. As a result, unreferenced mutable memory is not necessarily freed immediately; if it is acyclic it is freed when the last reference to it drops, but if it is part of a reference cycle it will be freed when the GC collects it (or when the owning task terminates, at the latest).

  Mutable memory can point to immutable memory but not vice-versa. Doing so merely delays (to an undefined future time) the moment when the deterministic, top-down destruction sequence for the referenced immutable memory *starts*. In other words, the

immutable "leaves" of a mutable structure are released in a locally-predictable order, even if the "interior" of the mutable structure is released in an unpredictable order.

- Typestate system

  Every storage slot in Rust participates in not only a conventional structural static type system, describing the interpretation of memory in the slot, but also a *typestate* system. The static typestates of a program describe the set of *pure, dynamic predicates* that provably hold over some set of slots, at each point in the program's control flow graph. The static calculation of the typestates of a program is a dataflow problem, and handles user-defined predicates in a similar fashion to the way the type system permits user-defined types.

  A short way of thinking of this is: types statically model the kinds of values held in slots, typestates statically model *assertions that hold* before and after statements.

- Static control over memory allocation, packing and aliasing.

  Every variable or field in Rust is a combination of a type, a mutability flag and a *mode*; this combination is called a *slot*. There are 3 kinds of *slot mode*, denoting 3 ways of referring to a value:

  - "interior" (slot contains value)
  - "exterior", (slot points to to managed heap allocation)
  - "alias", (slot points directly to provably-live address)

  Interior slots declared as variables in a function are allocated very quickly on the stack, as part of a local activation frame, as in C or C++. Alias slots permit efficient by-reference parameter passing without adjusting heap reference counts or interacting with garbage collection, as alias lifetimes are statically guaranteed to outlive callee lifetimes.

  Copying data between slots of different modes may cause either a simple address assignment or reference-count adjustment, or may cause a value to be "transplanted": copied by value from the interior of one memory structure to another, or between stack and heap. Transplanting, when necessary, is predictable and automatic, as part of the definition of the copy operator (`=`).

  In addition, slots have a static initialization state that is calculated by the typestate system. This permits late initialization of variables in functions with complex control-flow, while still guaranteeing that every use of a slot occurs after it has been initialized.

- Static control over mutability.

  Slots in Rust are classified as either immutable or mutable. By default, all slots are immutable.

  If a slot within a type is declared as `mutable`, the type is a `state` type and must be declared as such.

  This classification of data types in Rust interacts with the memory allocation and transmission rules. In particular:

  - Only immutable (non-state) values can be sent over channels.
  - Only immutable (non-state) objects can have destructor functions.

State values are subject to local (per-task) garbage-collection. Garbage collection costs are therefore also task-local and do not interrupt or suspend other tasks.

Immutable values are reference-counted and have a deterministic destruction order: top-down, immediately upon release of the last live reference.

State values can refer to immutable values, but not vice-versa. Rust therefore encourages the programmer to write in a style that consists primarily of immutable types, but also permits limited, local (per-task) mutability.

2.3 Influences

> The essential problem that must be solved in making a fault-tolerant software system is therefore that of fault-isolation. Different programmers will write different modules, some modules will be correct, others will have errors. We do not want the errors in one module to adversely affect the behaviour of a module which does not have any errors.
>
> - Joe Armstrong

> In our approach, all data is private to some process, and processes can only communicate through communications channels. *Security*, as used in this paper, is the property which guarantees that processes in a system cannot affect each other except by explicit communication.
>
> When security is absent, nothing which can be proven about a single module in isolation can be guaranteed to hold when that module is embedded in a system [...]
>
> - Robert Strom and Shaula Yemini

> Concurrent and applicative programming complement each other. The ability to send messages on channels provides I/O without side effects, while the avoidance of shared data helps keep concurrent processes from colliding.
>
> - Rob Pike

Rust is not a particularly original language. It may however appear unusual by contemporary standards, as its design elements are drawn from a number of "historical" languages that have, with a few exceptions, fallen out of favour. Five prominent lineages contribute the most:

- The NIL (1981) and Hermes (1990) family. These languages were developed by Robert Strom, Shaula Yemini, David Bacon and others in their group at IBM Watson Research Center (Yorktown Heights, NY, USA).

- The Erlang (1987) language, developed by Joe Armstrong, Robert Virding, Claes Wikström, Mike Williams and others in their group at the Ericsson Computer Science Laboratory (Älvsjö, Stockholm, Sweden) .

- The Sather (1990) language, developed by Stephen Omohundro, Chu-Cheow Lim, Heinz Schmidt and others in their group at The International Computer Science Institute of the University of California, Berkeley (Berkeley, CA, USA).

- The Newsqueak (1988), Alef (1995), and Limbo (1996) family. These languages were developed by Rob Pike, Phil Winterbottom, Sean Dorward and others in their group at Bell labs Computing Sciences Reserch Center (Murray Hill, NJ, USA).

- The Napier (1985) and Napier88 (1988) family. These languages were developed by Malcolm Atkinson, Ron Morrison and others in their group at the University of St. Andrews (St. Andrews, Fife, UK).

Additional specific influences can be seen from the following languages:

- The structural algebraic types and compilation manager of SML.
- The syntax-extension systems of Camlp4 and the Common Lisp readtable.
- The deterministic destructor system of C++.

3 Tutorial

TODO.

4 Reference

4.1 Ref.Lex

The lexical structure of a Rust source file or crate file is defined in terms of Unicode character codes and character properties.

Groups of Unicode character codes and characters are organized into *tokens*. Tokens are defined as the longest contiguous sequence of characters within the same token type (identifier, keyword, literal, symbol), or interrupted by ignored characters.

Most tokens in Rust follow rules similar to the C family.

Most tokens (including identifiers, whitespace, keywords, operators and structural symbols) are drawn from the ASCII-compatible range of Unicode. String and character literals, however, may include the full range of Unicode characters.

TODO: formalize this section much more.

4.1.1 Ref.Lex.Ignore

The classes of *whitespace* and *comment* is ignored, and are not considered as tokens.

Whitespace is any of the following Unicode characters: U+0020 (space), U+0009 (tab, `'\t'`), U+000A (LF, `'\n'`), U+000D (CR, `'\r'`).

Comments are any sequence of Unicode characters beginning with U+002F U+002F (`//`) and extending to the next U+000a character, *excluding* cases in which such a sequence occurs within a string literal token or a syntactic extension token.

4.1.2 Ref.Lex.Ident

Identifiers follow the pattern of C identifiers: they begin with a *letter* or underscore character `_` (Unicode character U+005f), and continue with any combination of *letters*, *digits* and underscores, and must not be equal to any keyword. See Section 4.1.3 [Ref.Lex.Key], page 17.

A *letter* is a Unicode character in the ranges U+0061-U+007A and U+0041-U+005A (`a-z` and `A-Z`).

A *digit* is a Unicode character in the range U+0030-U0039 (`0-9`).

4.1.3 Ref.Lex.Key

The keywords are:

```
use         meta       syntax     mutable    native
mod         import     export     let        auto
io          state      unsafe     auth       with
bind        type       true       false
any         int        uint       char       bool
u8          u16        u32        u64        f32
i8          i16        i32        i64        f64
rec         tup        tag        vec        str
fn          iter       obj        as         drop
task        port       chan       flush      spawn
if          else       alt        case       in
do          while      break      cont       fail
log         note       claim      check      prove
for         each       ret        put        be
```

4.1.4 Ref.Lex.Num

TODO: describe numeric literals.

4.1.5 Ref.Lex.Text

TODO: describe string and character literals.

4.1.6 Ref.Lex.Syntax

Syntactic extensions are marked with the *pound* sigil `#` (U+0023), followed by a qualified name of a compile-time imported module item, an optional parenthesized list of *tokens*, and an optional brace-enclosed region of free-form text (with brace-matching and brace-escaping used to determine the limit of the region). See Section 4.4.3 [Ref.Comp.Syntax], page 28.

TODO: formalize those terms more.

4.1.7 Ref.Lex.Sym

The special symbols are:

```
@       _
#       :       .       ;       ,
[       ]       {       }       (       )
=       <-      <|      <+      ->
+       ++      +=      -       --      -=
*       /       %       *=      /=      %=
&       |       !       ~       ^
&=      |=      ^=      !=
>>      >>>     <<      <<=     >>=     >>>=
<       <=      ==      >=      >
&&      ||
```

4.2 Ref.Path

A *path* is a ubiquitous syntactic form in Rust that deserves special attention. A path denotes a slot or an item. See Section 4.5.3 [Ref.Mem.Slot], page 32. See Section 4.7 [Ref.Item], page 40. Every slot and item in a Rust crate has a *canonical path* that refers to it from the crate top-level, as well as a number of shorter *relative paths* that may also denote it in inner scopes of the crate. There is no way to define a slot or item without a canonical path within its crate (with the exception of the crate's implicit top-level module). Paths have meaning only within a specific crate. See Section 4.4.1 [Ref.Comp.Crate], page 25.

Paths consist of period-separated components. In the simplest form, path components are identifiers. See Section 4.1.2 [Ref.Lex.Ident], page 16.

Two examples of simple paths consisting of only identifier components:

```
x;
x.y.z;
```

Paths fall into two important categories: *names* and *lvals*.

A *name* denotes an item, and is statically resolved to its referent at compile time.

An *lval* denotes a slot, and is statically resolved to a sequence of memory operations and primitive (arithmetic) expressions required to load or store to the slot at compile time.

In some contexts, the Rust grammar accepts a general *path*, but a subsequent syntactic restriction requires the path to be an lval or a name. In other words: in some contexts an lval is required (for example, on the left hand side of the copy operator, see Section 4.10.3 [Ref.Stmt.Copy], page 80) and in other contexts a name is required (for example, as a type parameter, see Section 4.7 [Ref.Item], page 40). In no case is the grammar made ambiguous by accepting a general path and restricting allowed paths to names or lvals after parsing. These restrictions are noted in the grammar. See Section 4.3 [Ref.Gram], page 23.

A name component may include type parameters. Type parameters are denoted by square brackets. Square brackets are used *only* to denote type parameters in Rust. If a name component includes a type parameter, the type parameter must also resolve statically to a type in the environment of the name. Type parameters are only part of the names of items. See Section 4.7 [Ref.Item], page 40.

An example of a name with type parameters:

```
m.map[int,str];
```

An lval component may include an indexing operator. Index operators are enclosed in parentheses and can include any integral expression. Indexing operators can only be applied to vectors or strings, and imply a run-time bounds-check. See Section 4.8.9 [Ref.Type.Vec], page 57.

An example of an lval with a dynamic indexing operator:

```
x.y.(1 + v).z;
```

4.3 Ref.Gram

TODO: LL(1), it reads like C, Alef and bits of Napier; formalize here.

4.4 Ref.Comp

Rust is a *compiled* language. Its semantics are divided along a *phase distinction* between compile-time and run-time. Those semantic rules that have a *static interpretation* govern the success or failure of compilation. A program that fails to compile due to violation of a compile-time rule has no defined semantics at run-time; the compiler should halt with an error report, and produce no executable artifact.

The compilation model centres on artifacts called *crates*. Each compilation is directed towards a single crate in source form, and if successful produces a single crate in executable form.

4.4.1 Ref.Comp.Crate

A *crate* is a unit of compilation and linking, as well as versioning, distribution and runtime loading. Crates are defined by *crate source files*, which are a type of source file written in a special declarative language: *crate language*.[1] A crate source file describes:

- Metadata about the crate, such as author, name, version, and copyright.
- The source-file and directory modules that make up the crate.
- The set of syntax extensions to enable for the crate.
- Any external crates or native modules that the crate imports to its top level.
- The organization of the crate's internal namespace.
- The set of names exported from the crate.

A single crate source file may describe the compilation of a large number of Rust source files; it is compiled in its entirety, as a single indivisible unit. The compilation phase attempts to transform a single crate source file, and its referenced contents, into a single compiled crate. Crate source files and compiled crates have a 1:1 relationship.

The syntactic form of a crate is a sequence of *directives*, some of which have nested sub-directives.

A crate defines an implicit top-level anonymous module: within this module, all members of the crate have canonical path names. See Section 4.2 [Ref.Path], page 22. The `mod` directives within a crate file specify sub-modules to include in the crate: these are either directory modules, corresponding to directories in the filesystem of the compilation environment, or file modules, corresponding to Rust source files. The names given to such modules in `mod` directives become prefixes of the paths of items and slots defined within any included Rust source files.

The `use` directives within the crate specify *other crates* to scan for, locate, import into the crate's module namespace during compilation, and link against at runtime. Use directives may also occur independently in rust source files. These directives may specify loose or tight "matching criteria" for imported crates, depending on the preferences of the crate developer. In the simplest case, a `use` directive may only specify a symbolic name and leave the task of locating and binding an appropriate crate to a compile-time heuristic. In a more controlled case, a `use` directive may specify any metadata as matching criteria, such as a URI, an author name or version number, a checksum or even a cryptographic signature, in order to select an an appropriate imported crate. See Section 4.4.2 [Ref.Comp.Meta], page 27.

The compiled form of a crate is a loadable and executable object file full of machine code, in a standard loadable operating-system format such as ELF, PE or Mach-O. The loadable object contains extensive DWARF metadata, describing:

- Metadata required for type reflection.
- The publicly exported module structure of the crate.
- Any metadata about the crate, defined by `meta` directives.
- The crates to dynamically link with at run-time, with matching criteria derived from the same `use` directives that guided compile-time imports.

[1] A crate is somewhat analogous to an *assembly* in the ECMA-335 CLI model, a *library* in the SML/NJ Compilation Manager, a *unit* in the Owens and Flatt module system, or a *configuration* in Mesa.

The `syntax` directives of a crate are similar to the `use` directives, except they govern the syntax extension namespace (accessed through the syntax-extension sigil `#`, see Section 4.4.3 [Ref.Comp.Syntax], page 28) available only at compile time. A `syntax` directive also makes its extension available to all subsequent directives in the crate file.

An example of a crate:

```
// Metadata about this crate
meta (author = "Jane Doe",
      name = "projx"
      desc = "Project X",
      ver = "2.5");

// Import a module.
use std (ver = "1.0");

// Activate a syntax-extension.
syntax re;

// Define some modules.
mod foo = "foo.rs";
mod bar {
    mod quux = "quux.rs";
}
```

4.4.2 Ref.Comp.Meta

In a crate, a `meta` directive associates free form key-value metadata with the crate. This metadata can, in turn, be used in providing partial matching parameters to syntax-extension loading and crate importing directives, denoted by `syntax` and `use` keywords respectively.

Alternatively, metadata can serve as a simple form of documentation.

4.4.3 Ref.Comp.Syntax

Rust provides a notation for *syntax extension*. The notation is a marked syntactic form that can appear as an expression, statement or item in the body of a Rust program, or as a directive in a Rust crate, and which causes the text enclosed within the marked form to be translated through a named extension function loaded into the compiler at compile-time.

The compile-time extension function must return a value of the corresponding Rust AST type, either an expression node, a statement node or an item node.[1] See Section 4.1.6 [Ref.Lex.Syntax], page 20.

A syntax extension is enabled by a `syntax` directive, which must occur in a crate file. When the Rust compiler encounters a `syntax` directive in a crate file, it immediately loads the named syntax extension, and makes it available for all subsequent crate directives within the enclosing block scope of the crate file, and all Rust source files referenced as modules from the enclosing block scope of the crate file.

For example, this extension might provide a syntax for regular expression literals:

```
// In a crate file:

// Requests the 're' syntax extension from the compilation environment.
syntax re;

// Also declares an import dependency on the module 're'.
use re;

// Reference to a Rust source file as a module in the crate.
mod foo = "foo.rs";

...

// In the source file "foo.rs", use the #re syntax extension and
// the re module at run-time.
let str s = get_string();
let regex pattern = #re.pat{ aa+b? };
let bool matched = re.match(pattern, s);
```

[1] The syntax-extension system is analogous to the extensible reader system provided by Lisp *readtables*, or the Camlp4 system of Objective Caml.

4.5 Ref.Mem

A Rust task's memory consists of a static set of *items*, a set of tasks each with its own *stack*, and a *heap*. Immutable portions of the heap may be shared between tasks, mutable portions may not.

Allocations in the stack and the heap consist of *slots*.

4.5.1 Ref.Mem.Alloc

The *items* of a program are those functions, iterators, objects, modules and types that have their value calculated at compile-time and stored uniquely in the memory image of the rust process. Items are neither dynamically allocated nor freed.

A task's *stack* consists of activation frames automatically allocated on entry to each function as the task executes. A stack allocation is reclaimed when control leaves the frame containing it.

The *heap* is a general term that describes two separate sets of exterior allocations: *local heap* allocations and the *shared heap* allocations.

Exterior allocations of mutable types are *local heap* allocations, owned by the task. Such *local allocations* cannot pass over channels and do not outlive the task that owns them. When unreferenced, they are collected using a general (cycle-aware) garbage-collector local to each task. Garbage collection within a local heap does not interrupt execution of other tasks.

Exterior allocations of immutable types are *shared heap* allocations, and can be multiply-referenced by many different tasks. Such *shared allocations* can pass over channels, and live as long as the last task referencing them. When unreferenced, they are collected immediately using reference-counting.

4.5.2 Ref.Mem.Own

A task *owns* all the interior allocations in its stack and *local* exterior allocations. A task *shares* ownership of *shared* exterior allocations. A task does not own any items.

Ownership of an allocation means that the owning task is the only task that can access the allocation.

Sharing of an allocation means that the same allocation may be concurrently referenced by multiple tasks. The only shared allocations are those that are immutable.

When a stack frame is exited, its interior allocations are all released, and its references to heap allocations (both shared and owned) are dropped.

When a task finishes, its stack is necessarily empty. The task's interior slots are released as the task itself is released, and its references to heap allocations are dropped.

4.5.3 Ref.Mem.Slot

A *slot* is a component of an allocation. A slot either holds a value or the address of another allocation. Every slot has one of three possible *modes*.

The possible *modes* of a slot are:

- *Interior mode*

  The slot holds the value of the slot.

- *Exterior mode*

  The slot holds the address of a heap allocation that holds the value of the slot.

  Exterior slots are indicated by the *at* sigil `@`.

  For example, the following code allocates an exterior record, copies it by counted-reference to a second exterior slot, then modifies the record through the second exterior slot that points to the same exterior allocation.

  ```
  type point3d = rec(int x, int y, int z);
  let @point3d pt1 = rec(x=1, y=2, z=3);
  let @point3d pt2 = pt1;
  pt2.z = 4;
  ```

- *Alias mode*

  The slot holds the address of a value. The referenced value may reside within a stack allocation *or* a heap allocation.

  Alias slots can *only* be declared as members of a function or iterator signature, bound to the lifetime of a stack frame. Alias slots cannot be declared as members of general data types.

  Alias slots are indicated by the *ampersand* sigil `&`.

  The following example function accepts a single read-only alias parameter:

  ```
  type point3d = rec(int x, int y, int z);

  fn extract_z(&point3d p) -> int {
      ret p.z;
  }
  ```

  The following example function accepts a single mutable alias parameter:

  ```
  fn incr(mutable &int i) {
      i = i + 1;
  }
  ```

4.5.4 Ref.Mem.Init

A slot is either initialized or uninitialized at every point in a program. An *initialized* slot is one that holds a value. An *uninitialized* slot is one that has not yet had a value written into it, or has had its value deleted, and so holds undefined memory. The typestate system ensures that an uninitialized slot cannot be read, but can be written to. A slot becomes initialized in any statement that writes to it, and remains initialized until explicitly destroyed or until its enclosing allocation is destroyed.

4.5.5 Ref.Mem.Acct

Every task belongs to a domain, and that domain tracks the amount of memory allocated and not yet released by tasks within it. See Section 4.6.3 [Ref.Task.Dom], page 38. Each domain has a memory budget. The *budget* of a domain is the maximum amount of memory that can be simultaneously allocated in the domain. If a task tries to allocate memory within a domain with an exceeded budget, the task will receive a signal.

Within a task, accounting is strictly enforced: all memory allocated through the runtime library, both user data, sub-domains and runtime-support structures such as channel and signal queues, are charged to a task's domain.

When a communication channel crosses from one domain to another, any value sent over the channel is guaranteed to have been *detached* from the domain's memory graph (singly referenced, and/or deep-copied), so its memory cost is transferred to the receiving domain.

4.6 Ref.Task

A executing Rust program consists of a tree of tasks. A Rust *task* consists of an entry function, a stack, a set of outgoing communication channels and incoming communication ports, and ownership of some portion of the heap of a single operating-system process.

Multiple Rust tasks may coexist in a single operating-system process. Execution of multiple Rust tasks in a single operating-system process may be either truly concurrent or interleaved by the runtime scheduler. Rust tasks are lightweight: each consumes less memory than an operating-system process, and switching between Rust tasks is faster than switching between operating-system processes.

4.6.1 Ref.Task.Comm

With the exception of *unsafe* constructs, Rust tasks are isolated from interfering with one another's memory directly. Instead of manipulating shared storage, Rust tasks communicate with one another using a typed, asynchronous, simplex message-passing system.

A *port* is a communication endpoint that can *receive* messages. Ports receive messages from channels.

A *channel* is a communication endpoint that can *send* messages. Channels send messages to ports.

Each port has a unique identity and cannot be replicated. If a port value is copied from one slot to another, both slots refer to the *same* port, even if the slots are declared as interior-mode. New ports can be constructed dynamically and stored in data structures.

Each channel is bound to a port when the channel is constructed, so the destination port for a channel must exist before the channel itself. A channel cannot be rebound to a different port from the one it was constructed with.

Many channels can be bound to the same port, but each channel is bound to a single port. In other words, channels and ports exist in an N:1 relationship, N channels to 1 port.[1]

Each port and channel can carry only one type of message. The message type is encoded as a parameter of the channel or port type. The message type of a channel is equal to the message type of the port it is bound to.

Messages are sent asynchronously or semi-synchronously. A channel contains a message queue and asynchronously sending a message merely inserts it into the channel's queue; message receipt is the responsibility of the receiving task.

Queued messages in channels are charged to the domain of the *sending* task. If too many messages are queued for transmission from a single sending task, without being received by a receiving task, the sending task may exceed its memory budget, which causes a run-time signal. To help control this possibility, a semi-synchronous send operation is possible, which blocks until there is room in the existing queue and then executes an asynchronous send. A full `flush` operation is also available, which blocks until a channel's queue is *empty*. A `flush` does *not* guarantee that a message has been *received* by any particular recipient when the sending task is unblocked. See Section 4.10.6 [Ref.Stmt.Flush], page 83.

The asynchronous message-send operator is `<+`. The semi-synchronous message-send operator is `<|`. See Section 4.10.5 [Ref.Stmt.Send], page 82. The message-receive operator is `<-`. See Section 4.10.7 [Ref.Stmt.Recv], page 84.

[1] It may help to remember nautical terminology when differentiating channels from ports. Many different waterways – channels – may lead to the same port.

4.6.2 Ref.Task.Life

The *lifecycle* of a task consists of a finite set of states and events that cause transitions between the states. The lifecycle states of a task are:

- running
- blocked
- failing
- dead

A task begins its lifecycle – once it has been spawned – in the *running* state. In this state it executes the statements of its entry function, and any functions called by the entry function.

A task may transition from the *running* state to the *blocked* state any time it executes a communication statement on a port or channel that cannot be immediately completed. When the communication statement can be completed – when a message arrives at a sender, or a queue drains sufficiently to complete a semi-synchronous send – then the blocked task will unblock and transition back to *running*.

A task may transition to the *failing* state at any time, due to an un-trapped signal or the execution of a `fail` statement. Once *failing*, a task unwinds its stack and transitions to the *dead* state. Unwinding the stack of a task is done by the task itself, on its own control stack. If a value with a destructor is freed during unwinding, the code for the destructor is run, also on the task's control stack. If the destructor code causes any subsequent state transitions, the task of unwinding and failing may suspend temporarily, and may involve (recursive) unwinding of the stack of a failed destructor. Nonetheless, the outermost unwinding activity will continue until the stack is unwound and the task transitions to the *dead* state. There is no way to "recover" from task failure.

A task in the *dead* state cannot transition to other states; it exists only to have its termination status inspected by other tasks, and/or to await reclamation when the last reference to it drops.

4.6.3 Ref.Task.Dom

Every task belongs to a domain. A *domain* is a structure that owns tasks, schedules tasks, tracks memory allocation within tasks and manages access to runtime services on behalf of tasks.

Typically each domain runs on a separate operating-system *thread*, or within an isolated operating-system *process*. An easy way to think of a domain is as an abstraction over either an operating-system thread *or* a process.

The key feature of a domain is that it isolates memory references created by the Rust tasks within it. No Rust task can refer directly to memory outside its domain.

Tasks can own sub-domains, which in turn own their own tasks. Every domain owns one *root task*, which is the root of the tree of tasks owned by the domain.

4.6.4 Ref.Task.Sched

Every task is *scheduled* within its domain. See Section 4.6.3 [Ref.Task.Dom], page 38. The currently scheduled task is given a finite *time slice* in which to execute, after which it is *descheduled* at a loop-edge or similar preemption point, and another task within the domain is scheduled, pseudo-randomly.

An executing task can `yield` control at any time, which deschedules it immediately. Entering any other non-executing state (blocked, dead) similarly deschedules the task.

4.7 Ref.Item

An *item* is a component of a module. Items are entirely determined at compile-time, remain constant during execution, and may reside in read-only memory.

There are 5 primary kinds of item: modules, functions, iterators, objects and types.

All items form an implicit scope for the declaration of sub-items. In other words, within a function, object or iterator, declarations of items can (in many cases) be mixed with the statements, control blocks, and similar artifacts that otherwise compose the item body. The meaning of these scoped items is the same as if the item was declared outside the scope, except that the item's *path name* within the module namespace is qualified by the name of the enclosing item. The exact locations in which sub-items may be declared is given by the grammar. See Section 4.3 [Ref.Gram], page 23.

Functions, iterators, objects and types may be *parametrized* by type. Type parameters are given as a comma-separated list of identifiers enclosed in square brackets (`[]`), after the name of the item and before its definition. The type parameters of an item are part of the name, not the type of the item; in order to refer to the type-parametrized item, a referencing name must in general provide type arguments as a list of comma-separated types enclosed within square brackets (though the type-inference system can often infer such argument types from context). There are no general parametric types.

4.7.1 Ref.Item.Mod

A *module item* contains declarations of other *items*. The items within a module may be functions, modules, objects or types. These declarations have both static and dynamic interpretation. The purpose of a module is to organize *names* and control *visibility*. Modules are declared with the keyword `mod`.

An example of a module:

```
mod math {
    type complex = (f64,f64);
    fn sin(f64) -> f64 {
        ...
    }
    fn cos(f64) -> f64 {
        ...
    }
    fn tan(f64) -> f64 {
        ...
    }
    ...
}
```

Modules may also include any number of *import and export declarations*. These declarations must precede any module item declarations within the module, and control the visibility of names both within the module and outside of it.

4.7.1.1 Ref.Item.Mod.Import

An *import declaration* creates one or more local name bindings synonymous with some other name. Usually an import declaration is used to shorten the path required to refer to a module item.

Note: unlike many languages, Rust's `import` declarations do *not* declare linkage-dependency with external crates. Linkage dependencies are independently declared with `use` declarations. See Section 4.4.1 [Ref.Comp.Crate], page 25.

An example of an import:

```
import std.math.sin;
fn main() {
    // Equivalent to 'log std.math.sin(1.0);'
    log sin(1.0);
}
```

4.7.1.2 Ref.Item.Mod.Export

An *export declaration* restricts the set of local declarations within a module that can be accessed from code outside the module. By default, all local declarations in a module are exported. If a module contains an export declaration, this declaration replaces the default export with the export specified.

An example of an export:

```
mod foo {
    export primary;

    fn primary() {
        helper(1, 2);
        helper(3, 4);
    }

    fn helper(int x, int y) {
        ...
    }
}

fn main() {
    foo.primary();  // Will compile.
    foo.helper(2,3) // ERROR: will not compile.
}
```

4.7.2 Ref.Item.Fn

A *function item* defines a sequence of statements associated with a name and a set of parameters. Functions are declared with the keyword `fn`. Functions declare a set of *input slots* as parameters, through which the caller passes arguments into the function, and an *output slot* through which the function passes results back to the caller.

A function may also be copied into a first class *value*, in which case the value has the corresponding *function type*, and can be used otherwise exactly as a function item (with a minor additional cost of calling the function, as such a call is indirect). See Section 4.8.11 [Ref.Type.Fn], page 59.

Every control path in a function ends with either a `ret` or `be` statement. If a control path lacks a `ret` statement in source code, an implicit `ret` statement is appended to the end of the control path during compilation, returning the implicit `()` value.

A function may have an *effect*, which may be either `io`, `state`, `unsafe`. If no effect is specified, the function is said to be *pure*.

Any pure boolean function is also called a *predicate*, and may be used as part of the static typestate system. See Section 4.10.1.3 [Ref.Stmt.Stat.Constr], page 73.

An example of a function:

```
fn add(int x, int y) -> int {
    ret x + y;
}
```

4.7.3 Ref.Item.Iter

Iterators are function-like items that can `put` multiple values during their execution before returning or tail-calling.

Putting a value is similar to returning a value – the argument to `put` is copied into the caller's frame and control transfers back to the caller – but the iterator frame is only *suspended* during the put, and will be *resumed* at the statement after the `put`, on the next iteration of the caller's loop.

The output type of an iterator is the type of value that the function will `put`, before it eventually executes a `ret` or `be` statement of type `()` and completes its execution.

An iterator can only be called in the loop header of a matching `for each` loop or as the argument in a `put each` statement. See Section 4.10.20 [Ref.Stmt.Foreach], page 97.

An example of an iterator:

```
iter range(int lo, int hi) -> int {
    let int i = lo;
    while (i < hi) {
        put i;
        i = i + 1;
    }
}

let int sum = 0;
for each (int x = range(0,100)) {
    sum += x;
}
```

4.7.4 Ref.Item.Obj

An *object item* defines the *state* and *methods* of a set of *object values*. Object values have object types. See Section 4.8.16 [Ref.Type.Obj], page 64.

An *object item* declaration – in addition to providing a scope for state and method declarations – implicitly declares a static function called the *object constructor*, as well as a named *object type*. The name given to the object item is resolved to a type when used in type context, or a constructor function when used in value context (such as a call).

Example of an object item:

```
obj counter(int state) {
    fn incr() {
       state += 1;
    }
    fn get() -> int {
       ret state;
    }
}

let counter c = counter(1);

c.incr();
c.incr();
check (c.get() == 3);
```

4.7.5 Ref.Item.Type

A *type* defines an *interpretation* of a value in memory. See Section 4.8 [Ref.Type], page 48. Types are declared with the keyword `type`. A type's interpretation is used for the values held in any slot with that type. See Section 4.5.3 [Ref.Mem.Slot], page 32. The interpretation of a value includes:

- Whether the value is composed of sub-values or is indivisible.
- Whether the value represents textual or numerical information.
- Whether the value represents integral or floating-point information.
- The sequence of memory operations required to access the value.
- Whether the value is mutable or immutable.

For example, the type `rec(u8 x, u8 y)` defines the interpretation of values that are composite records, each containing two unsigned two's complement 8-bit integers accessed through the components `x` and `y`, and laid out in memory with the `x` component preceding the `y` component.

Some types are *recursive*. A recursive type is one that includes its own definition as a component, by named reference. Recursive types are restricted to occur only within a single crate, and only through a restricted form of `tag` type. See Section 4.8.10 [Ref.Type.Tag], page 58.

4.8 Ref.Type

Every slot and value in a Rust program has a type. The *type* of a *value* defines the interpretation of the memory holding it. The type of a *slot* may also include constraints. See Section 4.8.17 [Ref.Type.Constr], page 66.

Built-in types and type-constructors are tightly integrated into the language, in non-trivial ways that are not possible to emulate in user-defined types. User-defined types have limited capabilities. In addition, every built-in type or type-constructor name is reserved as a *keyword* in Rust; they cannot be used as user-defined identifiers in any context.

4.8.1 Ref.Type.Any

The type `any` is the union of all possible Rust types. A value of type `any` is represented in memory as a pair consisting of an exterior value of some non-`any` type T and a reflection of the type T.

Values of type `any` can be used in an `alt type` statement, in which the reflection is used to select a block corresponding to a particular type extraction. See Section 4.10.22 [Ref.Stmt.Alt], page 99.

4.8.2 Ref.Type.Mach

The machine types are the following:

- The unsigned two's complement word types `u8`, `u16`, `u32` and `u64`, with values drawn from the integer intervals $[0, 2^8 - 1]$, $[0, 2^{16} - 1]$, $[0, 2^{32} - 1]$ and $[0, 2^{64} - 1]$ respectively.
- The signed two's complement word types `i8`, `i16`, `i32` and `i64`, with values drawn from the integer intervals $[-(2^7), (2^7) - 1)]$, $[-(2^{15}), 2^{15} - 1)]$, $[-(2^{31}), 2^{31} - 1)]$ and $[-(2^{63}), 2^{63} - 1)]$ respectively.
- The IEEE 754 single-precision and double-precision floating point types: `f32` and `f64`, respectively.

4.8.3 Ref.Type.Int

The Rust type `uint`[1] is a two's complement unsigned integer type with with target-machine-dependent size. Its size, in bits, is equal to the number of bits required to hold any memory address on the target machine.

The Rust type `int`[2] is a two's complement signed integer type with target-machine-dependent size. Its size, in bits, is equal to the size of the rust type `uint` on the same target machine.

[1] A Rust `uint` is analogous to a C99 `uintptr_t`.

[2] A Rust `int` is analogous to a C99 `intptr_t`.

4.8.4 Ref.Type.Prim

The primitive types are the following:

- The "nil" type `()`, having the single "nil" value `()`.[1]
- The boolean type `bool` with values `true` and `false`.
- The machine types.
- The machine-dependent integer types.

[1] The "nil" value `()` is *not* a sentinel "null pointer" value for alias or exterior slots; the "nil" type is the implicit return type from functions otherwise lacking a return type, and can be used in other contexts (such as message-sending or type-parametric code) as a zero-byte type.

4.8.5 Ref.Type.Big

The Rust type `big`[1] is an arbitrary precision integer type that fits in a machine word *when possible* and transparently expands to a boxed "big integer" allocated in the run-time heap when it overflows or underflows outside of the range of a machine word.

A Rust `big` grows to accommodate extra binary digits as they are needed, by taking extra memory from the memory budget available to each Rust task, and should only exhaust its range due to memory exhaustion.

[1] A Rust `big` is analogous to a Lisp bignum or a Python long integer.

4.8.6 Ref.Type.Text

The types `char` and `str` hold textual data.

A value of type `char` is a Unicode character, represented as a 32-bit unsigned word holding a UCS-4 codepoint.

A value of type `str` is a Unicode string, represented as a vector of 8-bit unsigned bytes holding a sequence of UTF-8 codepoints.

4.8.7 Ref.Type.Rec

The record type-constructor `rec` forms a new heterogeneous product of slots.[1] Fields of a `rec` type are accessed by name and are arranged in memory in the order specified by the `rec` type.

An example of a `rec` type and its use:

```
type point = rec(int x, int y);
let point p = rec(x=10, y=11);
let int px = p.x;
```

[1] The `rec` type-constructor is analogous to the `struct` type-constructor in the Algol/C family, the *record* types of the ML family, or the *structure* types of the Lisp family.

4.8.8 Ref.Type.Tup

The tuple type-constructor `tup` forms a new heterogeneous product of slots exactly as the `rec` type-constructor does, with the difference that tuple slots are automatically assigned implicit field names, given by ascending integers prefixed by the underscore character: `_0`, `_1`, `_2`, etc. The fields of a tuple are laid out in memory contiguously, like a record, in order specified by the tuple type.

An example of a tuple type and its use:

```
type pair = tup(int,str);
let pair p = tup(10,"hello");
check (p._0 == 10);
p._1 = "world";
check (p._1 == "world");
```

4.8.9 Ref.Type.Vec

The vector type-constructor `vec` represents a homogeneous array of slots. A vector has a fixed size, and may or may not have mutable member slots. If the slots of a vector are mutable, the vector is a *state* type.

Vectors can be sliced. A slice expression builds a new vector by copying a contiguous range – given by a pair of indices representing a half-open interval – out of the sliced vector.

And example of a `vec` type and its use:

```
let vec[int] v = vec(7, 5, 3);
let int i = v.(2);
let vec[int] v2 = v.(0,1); // Form a slice.
```

Vectors always *allocate* a storage region sufficient to store the first power of two worth of elements greater than or equal to the size of the largest slice sharing the storage. This behaviour supports idiomatic in-place "growth" of a mutable slot holding a vector:

```
let mutable vec[int] v = vec(1, 2, 3);
v += vec(4, 5, 6);
```

Normal vector concatenation causes the allocation of a fresh vector to hold the result; in this case, however, the slot holding the vector recycles the underlying storage in-place (since the reference-count of the underlying storage is equal to 1).

All accessible elements of a vector are always initialized, and access to a vector is always bounds-checked.

4.8.10 Ref.Type.Tag

The `tag` type-constructor forms new heterogeneous disjoint sum types.[1] A `tag` type consists of a number of *variants*, each of which is independently named and takes an optional tuple of arguments.

The variants of a `tag` type may be recursive: that is, the definition of a `tag` type may refer to type definitions that include the defined `tag` type itself. Such recursion has restrictions:

- Recursive types can only be introduced through `tag` types.
- A recursive `tag` type must have at least one non-recursive variant (in order to give the recursion a basis case).
- The recursive slots of recursive variants must be *exterior* slots (in order to bound the in-memory size of the variant).
- Recursive type definitions can cross module boundaries, but not module *visibility* boundaries, nor crate boundaries (in order to simplify the module system).

An example of a `tag` type and its use:

```
type animal = tag(dog, cat);
let animal a = dog;
a = cat;
```

An example of a *recursive* `tag` type and its use:

```
type list[T] = tag(nil,
                   cons(T, @list[T]));
let list[int] a = cons(7, cons(13, nil));
```

[1] The `tag` type is analogous to a `data` constructor declaration in ML or a *pick ADT* in Limbo.

4.8.11 Ref.Type.Fn

The function type-constructor `fn` forms new function types. A function type consists of a sequence of input slots, an optional set of input constraints (see Section 4.10.1.3 [Ref.Stmt.Stat.Constr], page 73), an output slot, and an *effect*. See Section 4.7.2 [Ref.Item.Fn], page 44.

An example of a `fn` type:

```
fn add(int x, int y) -> int {
  ret x + y;
}

let int x = add(5,7);

type binop = fn(int,int) -> int;
let binop bo = add;
x = bo(5,7);
```

4.8.12 Ref.Type.Iter

The iterator type-constructor `iter` forms new iterator types. An iterator type consists a sequence of input slots, an optional set of input constraints, an output slot, and an *effect*. See Section 4.7.3 [Ref.Item.Iter], page 45.

An example of an `iter` type:

```
iter range(int x, int y) -> int {
  while (x < y) {
    put x;
    x += 1;
  }
}

for each (int i = range(5,7)) {
  ...;
}
```

4.8.13 Ref.Type.Port

The port type-constructor `port` forms types that describe ports. A port is the *receiving end* of a typed, asynchronous, simplex inter-task communication facility. See Section 4.6.1 [Ref.Task.Comm], page 36. A `port` type takes a single type parameter, denoting the type of value that can be received from a `port` value of that type.

Ports are modeled as mutable native types with built-in meaning to the language. They cannot be transmitted over channels or otherwise replicated, and are always local to the task that creates them.

An example of a `port` type:

```
type port[vec[str]] svp;
let svp p = get_port();
let vec[str] v;
v <- p;
```

4.8.14 Ref.Type.Chan

The channel type-constructor `chan` forms types that describe channels. A channel is the *sending end* of a typed, asynchronous, simplex inter-task communication facility. See Section 4.6.1 [Ref.Task.Comm], page 36. A `chan` type takes a single type parameter, denoting the type of value that can be sent to a channel of that type.

Channels are immutable, and can be transmitted over channels to other tasks. They are modeled as immutable native types with built-in meaning to the language.

When a task sends a message into a channel, the task forms an outgoing queue associated with that channel. The per-task queue *associated* with a channel can be indirectly manipulated by the task, but is *not* otherwise considered "part of" to the channel: the queue is "part of" the *sending task*. Sending a channel to another task does not copy the queue associated with the channel.

Channels are also *weak*: a channel is directly coupled to a particular destination port on a particular task, but does not keep that port or task *alive*. A channel may therefore fail to operate at any moment. If a task sends to a channel that is connected to a nonexistent port, it receives a signal.

An example of a `chan` type:

```
type chan[vec[str]] svc;
let svc c = get_chan();
let vec[str] v = vec("hello", "world");
c <| v;
```

4.8.15 Ref.Type.Task

The task type `task` describes values that are *live tasks*.

Tasks form an *ownership tree* in which each task (except the root task) is directly owned by exactly one parent task. The purpose of a variable of `task` type is to manage the lifecycle of the associated task. Communication is carried out solely using channels and ports.

Like ports, tasks are modeled as mutable native types with built-in meaning to the language. They cannot be transmitted over channels or otherwise replicated, and are always local to the task that spawns them.

If all references to a task are dropped (due to the release of any slots holding those references), the released task immediately fails. See Section 4.6.2 [Ref.Task.Life], page 37.

4.8.16 Ref.Type.Obj

A *object type* describes values of abstract type, that carry some hidden *fields* and are accessed through a set of un-ordered *methods*. Every object item (see Section 4.7.4 [Ref.Item.Obj], page 46) implicitly declares an object type carrying methods with types derived from all the methods of the object item.

Object types can also be declared in isolation, independent of any object item declaration. Such a "plain" object type can be used to describe an interface that a variety of particular objects may conform to, by supporting a superset of the methods.

An object type that can contain a state must be declared as a `state obj` like any other state type. And similarly a method type that performs I/O or makes native calls must be declared `io` or `unsafe`, like any other function.

Moreover, *all* methods of a state object are implicitly state functions – as they all bind the same mutable state field(s) – so implicitly have an effect lower than `io`. It is therefore unnecessary to declare methods within a state object type (or state object item) as `io`.

An example of an object type with two separate object items supporting it, and a client function using both items via the object type:

```
state type taker =
    state obj {
        fn take(int);
    };

state obj adder(mutable int x) {
    fn take(int y) {
        x += y;
    }
}

obj sender(chan[int] c) {
    io fn take(int z) {
        c <| z;
    }
}

fn give_ints(taker t) {
    t.take(1);
    t.take(2);
    t.take(3);
}

let port[int] p = port();

let taker t1 = adder(0);
let taker t2 = sender(chan(p));
```

```
give_ints(t1);
give_ints(t2);
```

4.8.17 Ref.Type.Constr

A *constrained type* is a type that carries a *formal constraint* (see Section 4.10.1.3 [Ref.Stmt.Stat.Constr], page 73), which is similar to a normal constraint except that the *base name* of any slots mentioned in the constraint must be the special *formal symbol* *.

When a constrained type is instantiated in a particular slot declaration, the formal symbol in the constraint is replaced with the name of the declared slot and the resulting constraint is checked immediately after the slot is declared. See Section 4.10.24 [Ref.Stmt.Check], page 104.

An example of a constrained type with two separate instantiations:

```
type ordered_range = rec(int low, int high) : less_than(*.low, *.high);

let ordered_range rng1 = rec(low=5, high=7);
// implicit: 'check less_than(rng1.low, rng1.high);'

let ordered_range rng2 = rec(low=15, high=17);
// implicit: 'check less_than(rng2.low, rng2.high);'
```

4.8.18 Ref.Type.Type

TODO.

4.9 Ref.Expr

Rust has two kinds of expressions: *parsed expressions* and *primitive expressions*. The former are syntactic sugar and are eliminated during parsing. The latter are very minimal, consisting only of paths and primitive literals, possibly combined via a single level (non-recursive) unary or binary machine-level operation (ALU or FPU). See Section 4.2 [Ref.Path], page 22.

For the most part, Rust semantics are defined in terms of *statements*, which parsed expressions are desugared to. The desugaring is defined in the grammar. See Section 4.3 [Ref.Gram], page 23. The residual primitive statements appear only in the right hand side of copy statements, See Section 4.10.3 [Ref.Stmt.Copy], page 80.

4.10 Ref.Stmt

A *statement* is a component of a block, which is in turn a components of an outer block, a function or an iterator. When a function is spawned into a task, the task *executes* statements in an order determined by the body of the enclosing structure. Each statement causes the task to perform certain actions.

4.10.1 Ref.Stmt.Stat

Statements have a detailed static semantics. The static semantics determine, on a statement-by-statement basis, the *effects* the statement has on its environment, as well the *legality* of the statement in its environment.

The legality of a statement is partly governed by syntactic rules, partly by its conformance to the types of slots it affects, and partly by a statement-oriented static dataflow analysis. This section describes the statement-oriented static dataflow analysis, also called the *typestate* system.

4.10.1.1 Ref.Stmt.Stat.Point

A *point* exists before and after any statement in a Rust program. For example, this code:

```
s = "hello, world";
print(s);
```

Consists of two statements and four points:

- the point before the first statement
- the point after the first statement
- the point before the second statement
- the point after the second statement

The typestate system reasons over points, rather than statements. This may seem counter-intuitive, but points are the more primitive concept. Another way of thinking about a point is as a set of *instants in time* at which the state of a task is fixed. By contrast, a statement represents a *duration in time*, during which the state of the task changes. The typestate system is concerned with constraining the possible states of a task's memory at *instants*; it is meaningless to speak of the state of a task's memory "at" a statement, as each statement is likely to change the contents of memory.

4.10.1.2 Ref.Stmt.Stat.CFG

Each *point* can be considered a vertex in a directed *graph*. Each kind of statement implies a single edge in this graph between the point before the statement and the point after it, as well as a set of zero or more edges from the points of the statement to points before other statements. The edges between points represent *possible* indivisible control transfers that might occur during execution.

This implicit graph is called the *control flow graph*, or *CFG*.

4.10.1.3 Ref.Stmt.Stat.Constr

A *predicate* is any pure boolean function. See Section 4.7.2 [Ref.Item.Fn], page 44.

A *constraint* is a predicate applied to specific slots.

For example, consider the following code:

```
fn is_less_than(int a, int b) -> bool {
     ret a < b;
}

fn test() {
   let int x = 10;
   let int y = 20;
   check is_less_than(x,y);
}
```

This example defines the predicate `is_less_than`, and applies it to the slots `x` and `y`. The constraint being checked on the third line of the function is `is_less_than(x,y)`.

Predicates can only apply to slots holding immutable values. The slots a predicate applies to can themselves be mutable, but the types of values held in those slots must be immutable.

4.10.1.4 Ref.Stmt.Stat.Cond

A *condition* is a set of zero or more constraints.

Each *point* has an associated *condition*:

- The *precondition* of a statement is the condition the statement requires in the point before the condition.
- The *postcondition* of a statement is the condition the statement enforces in the point after the statement.

Any constraint present in the precondition and *absent* in the postcondition is considered to be *dropped* by the statement.

4.10.1.5 Ref.Stmt.Stat.Typestate

The typestate checking system *calculates* an additional condition for each point called its typestate. For a given statement, we call the two typestates associated with its two points the prestate and a poststate.

- The *prestate* of a statement is the typestate of the point before the statement.
- The *poststate* of a statement is the typestate of the point after the statement.

A *typestate* is a condition that has *been determined by the typestate algorithm* to hold at a point. This is a subtle but important point to understand: preconditions and postconditions are *inputs* to the typestate algorithm; prestates and poststates are *outputs* from the typestate algorithm.

The typestate algorithm analyses the preconditions and postconditions of every statement in a block, and computes a condition for each typestate. Specifically:

- Initially, every typestate is empty.
- Each statement's poststate is given the union of the statement's prestate, precondition, and postcondition.
- Each statement's poststate has the difference between the statement's precondition and postcondition removed.
- Each statement's prestate is given the intersection of the poststates of every parent statement in the CFG.
- The previous three steps are repeated until no typestates in the block change.

The typestate algorithm is a very conventional dataflow calculation, and can be performed using bit-set operations, with one bit per predicate and one bit-set per condition.

After the typestates of a block are computed, the typestate algorithm checks that every constraint in the precondition of a statement is satisfied by its prestate. If any preconditions are not satisfied, the mismatch is considered a static (compile-time) error.

4.10.1.6 Ref.Stmt.Stat.Check

The key mechanism that connects run-time semantics and compile-time analysis of typestates is the use of `check` statements. See Section 4.10.24 [Ref.Stmt.Check], page 104. A `check` statement guarantees that *if* control were to proceed past it, the predicate associated with the `check` would have succeeded, so the constraint being checked *statically* holds in subsequent statements.[1]

It is important to understand that the typestate system has *no insight* into the meaning of a particular predicate. Predicates and constraints are not evaluated in any way at compile time. Predicates are treated as specific (but unknown) functions applied to specific (also unknown) slots. All the typestate system does is track which of those predicates – whatever they calculate – *must have been checked already* in order for program control to reach a particular point in the CFG. The fundamental building block, therefore, is the `check` statement, which tells the typestate system "if control passes this statement, the checked predicate holds".

From this building block, constraints can be propagated to function signatures and constrained types, and the responsibility to `check` a constraint pushed further and further away from the site at which the program requires it to hold in order to execute properly.

[1] A `check` statement is similar to an `assert` call in a C program, with the significant difference that the Rust compiler *tracks* the constraint that each `check` statement enforces. Naturally, `check` statements cannot be omitted from a "production build" of a Rust program the same way `asserts` are frequently disabled in deployed C programs.

4.10.2 Ref.Stmt.Decl

A *declaration statement* is one that introduces a *name* into the enclosing statement block. The declared name may denote a new slot or a new item. The scope of the name extends to the entire containing block, both before and after the declaration.

4.10.2.1 Ref.Stmt.Decl.Item

An *item declaration statement* has a syntactic form identical to an item declaration within a module. Declaring an item – a function, iterator, object, type or module – locally within a statement block is simply a way of restricting its scope to a narrow region containing all of its uses; it is otherwise identical in meaning to declaring the item outside the statement block.

Note: there is no implicit capture of the function's dynamic environment when declaring a function-local item.

4.10.2.2 Ref.Stmt.Decl.Slot

A `slot declaration statement` has one one of two forms:

- `let` *mode-and-type slot optional-init*;
- `auto` *slot optional-init*;

Where *mode-and-type* is a slot mode and type expression, *slot* is the name of the slot being declared, and *optional-init* is either the empty string or an equals sign (`=`) followed by a primitive expression.

Both forms introduce a new slot into the containing block scope. The new slot is visible across the entire scope, but is initialized only at the point following the declaration statement.

The latter (`auto`) form of slot declaration causes the compiler to infer the static type of the slot through unification with the types of values assigned to the slot in the the remaining code in the block scope. Inferred slots always have *interior* mode. See Section 4.5.3 [Ref.Mem.Slot], page 32.

4.10.3 Ref.Stmt.Copy

A *copy statement* consists of an *lval* – a name denoting a slot – followed by an equals-sign (`=`) and a primitive expression. See Section 4.9 [Ref.Expr], page 68.

Executing a copy statement causes the value denoted by the expression – either a value in a slot or a primitive combination of values held in slots – to be copied into the slot denoted by the *lval*.

A copy may entail the formation of references, the adjustment of reference counts, execution of destructors, or similar adjustments in order to respect the `lval` slot mode and any existing value held in it. All such adjustment is automatic and implied by the `=` operator.

An example of three different copy statements:

```
x = y;
x.y = z;
x.y = z + 2;
```

4.10.4 Ref.Stmt.Spawn

A `spawn` statement consists of keyword `spawn`, followed by a normal *call* statement (see Section 4.10.8 [Ref.Stmt.Call], page 85). A `spawn` statement causes the runtime to construct a new task executing the called function. The called function is referred to as the *entry function* for the spawned task, and its arguments are copied form the spawning task to the spawned task before the spawned task begins execution.

Only arguments of interior or exterior mode are permitted in the function called by a spawn statement, not arguments with alias mode.

The result of a `spawn` statement is a `task` value.

An example of a `spawn` statement:

```
fn helper(chan[u8] out) {
    // do some work.
    out <| result;
}

let port[u8] out;
let task p = spawn helper(chan(out));
// let task run, do other things.
auto result <- out;
```

4.10.5 Ref.Stmt.Send

Sending a value through a channel can be done via two different statements. Both statements take an *lval*, denoting a channel, and a value to send into the channel. The action of *sending* varies depending on the *send operator* employed.

The *asynchronous send* operator `<+` adds a value to the channel's queue, without blocking. If the queue is full, it is extended, taking memory from the task's domain. If the task memory budget is exhausted, a signal is sent to the task.

The *semi-synchronous send* operator `<|` adds a value to the channel's queue *only if* the queue has room; if the queue is full, the operation *blocks* the sender until the queue has room.

An example of an asynchronous send:

```
chan[str] c = ...;
c <+ "hello, world";
```

An example of a semi-synchronous send:

```
chan[str] c = ...;
c <| "hello, world";
```

4.10.6 Ref.Stmt.Flush

A `flush` statement takes a channel and blocks the flushing task until the channel's queue has emptied. It can be used to implement a more precise form of flow-control than with the send operators alone.

An example of the `flush` statement:

```
chan[str] c = ...;
c <| "hello, world";
flush c;
```

4.10.7 Ref.Stmt.Recv

The *receive statement* takes an *lval* to receive into and an expression denoting a port, and applies the *receive operator* (`<-`) to the pair, copying a value out of the port and into the *lval*. The statement causes the receiving task to enter the *blocked reading* state until a task is sending a value to the port, at which point the runtime pseudo-randomly selects a sending task and copies a value from the head of one of the task queues to the receiving slot, and un-blocks the receiving task. See Section 4.11.3 [Ref.Run.Comm], page 109.

An example of a *receive*:

```
port[str] p = ...;
let str s <- p;
```

4.10.8 Ref.Stmt.Call

A *call statement* invokes a function, providing a tuple of input slots and a reference to an output slot. If the function eventually returns, then the statement completes.

A call statement statically requires that the precondition declared in the callee's signature is satisfied by the statement prestate. In this way, typestates propagate through function boundaries. See Section 4.10.1 [Ref.Stmt.Stat], page 70.

An example of a call statement:

```
let int x = add(1, 2);
```

4.10.9 Ref.Stmt.Bind

A *bind statement* constructs a new function from an existing function.[1] The new function has zero or more of its arguments *bound* into a new, hidden exterior tuple that holds the bindings. For each concrete argument passed in the `bind` statement, the corresponding parameter in the existing function is *omitted* as a parameter of the new function. For each argument passed the placeholder symbol `_` in the `bind` statement, the corresponding parameter of the existing function is *retained* as a parameter of the new function.

Any subsequent invocation of the new function with residual arguments causes invocation of the existing function with the combination of bound arguments and residual arguments that was specified during the binding.

An example of a `bind` statement:

```
fn add(int x, int y) -> int {
    ret x + y;
}
type single_param_fn = fn(int) -> int;

let single_param_fn add4 = bind add(4, _);

let single_param_fn add5 = bind add(_, 5);

check (add(4,5) == add4(5));
check (add(4,5) == add5(4));
```

A `bind` statement generally stores a copy of the bound arguments in the hidden exterior tuple. For bound interior slots and alias slots in the bound function signature, an interior slot is allocated in the hidden tuple and populated with a copy of the bound value. For bound exterior slots in the bound function signature, an exterior slot is allocated in the hidden tuple and populated with a copy of the bound value, an exterior (pointer) value.

The `bind` statement is a lightweight mechanism for simulating the more elaborate construct of *lexical closures* that exist in other languages. Rust has no support for lexical closures, but many realistic uses of them can be achieved with `bind` statements.

[1] The `bind` statement is analogous to the `bind` expression in the Sather language.

4.10.10 Ref.Stmt.Ret

Executing a `ret` statement[1] copies a value into the return slot of the current function, destroys the current function activation frame, and transfers control to the caller frame.

An example of a `ret` statement:

```
fn max(int a, int b) -> int {
   if (a > b) {
      ret a;
   }
   ret b;
}
```

[1] A `ret` statement is analogous to a `return` statement in the C family.

4.10.11 Ref.Stmt.Be

Executing a `be` statement[1] destroys the current function activation frame and replaces it with an activation frame for the called function. In other words, `be` executes a tail-call. The syntactic form of a `be` statement is therefore limited to *tail position*: its argument must be a *call expression*, and it must be the last statement in a block.

An example of a `be` statement:

```
fn print_loop(int n) {
  if (n <= 0) {
    ret;
  } else {
    print_int(n);
    be print_loop(n-1);
  }
}
```

The above example executes in constant space, replacing each frame with a new copy of itself.

[1] A `be` statement in is analogous to a `become` statement in Newsqueak or Alef.

4.10.12 Ref.Stmt.Put

Executing a `put` statement copies a value into the put slot of the current iterator, suspends execution of the current iterator, and transfers control to the current put-recipient frame.

A `put` statement is only valid within an iterator.[1] The current put-recipient will eventually resume the suspended iterator containing the `put` statement, either continuing execution after the `put` statement, or terminating its execution and destroying the iterator frame.

[1] A `put` statement is analogous to a `yield` statement in the CLU, Sather and Objective C 2.0 languages, or in more recent languages providing a "generator" facility, such as Python, Javascript or C#. Like the generators of CLU, Sather and Objective C 2.0, but *unlike* these later languages, Rust's iterators reside on the stack and obey a strict stack discipline.

4.10.13 Ref.Stmt.Fail

Executing a `fail` statement causes a task to enter the *failing* state. In the *failing* state, a task unwinds its stack, destroying all frames and freeing all resources until it reaches its entry frame, at which point it halts execution in the *dead* state.

4.10.14 Ref.Stmt.Log

Executing a `log` statement may, depending on runtime configuration, cause a value to be appended to an internal diagnostic logging buffer provided by the runtime or emitted to a system console. Log statements are enabled or disabled dynamically at run-time on a per-task and per-item basis. See Section 4.11.5 [Ref.Run.Log], page 111.

Executing a `log` statement not considered an `io` effect in the effect system. In other words, a pure function remains pure even if it contains a log statement.

4.10.15 Ref.Stmt.Note

A `note` statement has no effect during normal execution. The purpose of a `note` statement is to provide additional diagnostic information to the logging subsystem during task failure. See Section 4.10.14 [Ref.Stmt.Log], page 91. Using `note` statements, normal diagnostic logging can be kept relatively sparse, while still providing verbose diagnostic "back-traces" when a task fails.

When a task is failing, control frames *unwind* from the innermost frame to the outermost, and from the innermost lexical block within an unwinding frame to the outermost. When unwinding a lexical block, the runtime processes all the `note` statements in the block sequentially, from the first statement of the block to the last. During processing, a `note` statement has equivalent meaning to a `log` statement: it causes the runtime to append the argument of the `note` to the internal logging diagnostic buffer.

An example of a `note` statement:

```
fn read_file_lines(&str path) -> vec[str] {
    note path;
    vec[str] r;
    file f = open_read(path);
    for* (str &s = lines(f)) {
        vec.append(r,s);
    }
    ret r;
}
```

In this example, if the task fails while attempting to open or read a file, the runtime will log the path name that was being read. If the function completes normally, the runtime will not log the path.

A slot that is marked by a `note` statement does *not* have its value copied aside when control passes through the `note`. In other words, if a `note` statement notes a particular slot, and code after the `note` that slot, and then a subsequent failure occurs, the *mutated* value will be logged during unwinding, *not* the original value that was held in the slot at the moment control passed through the `note` statement.

4.10.16 Ref.Stmt.While

A `while` statement is a loop construct. A `while` loop may be either a simple `while` or a `do-while` loop.

In the case of a simple `while`, the loop begins by evaluating the boolean loop conditional expression. If the loop conditional expression evaluates to `true`, the loop body block executes and control returns to the loop conditional expression. If the loop conditional expression evaluates to `false`, the `while` statement completes.

In the case of a `do-while`, the loop begins with an execution of the loop body. After the loop body executes, it evaluates the loop conditional expression. If it evaluates to `true`, control returns to the beginning of the loop body. If it evaluates to `false`, control exits the loop.

An example of a simple `while` statement:

```
while (i < 10) {
    print("hello\n");
    i = i + 1;
}
```

An example of a `do-while` statement:

```
do {
    print("hello\n");
    i = i + 1;
} while (i < 10);
```

4.10.17 Ref.Stmt.Break

Executing a `break` statement immediately terminates the innermost loop enclosing it. It is only permitted in the body of a loop.

4.10.18 Ref.Stmt.Cont

Executing a `cont` statement immediately terminates the current iteration of the innermost loop enclosing it, returning control to the loop *head*. In the case of a `while` loop, the head is the conditional expression controlling the loop. In the case of a `for` or `for each` loop, the head is the iterator or vector-slice increment controlling the loop.

A `cont` statement is only permitted in the body of a loop.

4.10.19 Ref.Stmt.For

A *for loop* is controlled by a vector or string. The for loop bounds-checks the underlying sequence *once* when initiating the loop, then repeatedly copies each value of the underlying sequence into the element variable, executing the loop body once per copy. To perform a for loop on a sub-range of a vector or string, form a temporary slice over the sub-range and run the loop over the slice.

Example of a 4 for loops, all identical:

```
let vec[foo] v = vec(a, b, c);

for (&foo e in v.(0, _vec.len(v))) {
    bar(e);
}

for (&foo e in v.(0,)) {
    bar(e);
}

for (&foo e in v.(,)) {
    bar(e);
}

for (&foo e in v) {
    bar(e);
}
```

4.10.20 Ref.Stmt.Foreach

An *foreach loop* is denoted by the `for each` keywords, and is controlled by an iterator. The loop executes once for each value `put` by the iterator. When the iterator returns or fails, the loop terminates.

Example of a foreach loop:

```
let str txt;
let vec[str] lines;
for each (&str s = _str.split(txt, "\n")) {
    vec.push(lines, s);
}
```

4.10.21 Ref.Stmt.If

An `if` statement is a conditional branch in program control. The form of an `if` statement is a parenthesized condition expression, followed by a consequent block, and an optional trailing `else` block. The condition expression must have type `bool`. If the condition expression evaluates to `true`, the consequent block is executed and any `else` block is skipped. If the condition expression evaluates to `false`, the consequent block is skipped and any `else` block is executed.

4.10.22 Ref.Stmt.Alt

An `alt` statement is a multi-directional branch in program control. There are three kinds of `alt` statement: communication `alt` statements, pattern `alt` statements, and `alt type` statements.

The form of each kind of `alt` is similar: an initial *head* that describes the criteria for branching, followed by a sequence of zero or more *arms*, each of which describes a *case* and provides a *block* of statements associated with the case. When an `alt` is executed, control enters the head, determines which of the cases to branch to, branches to the block associated with the chosen case, and then proceeds to the statement following the `alt` when the case block completes.

4.10.22.1 Ref.Stmt.Alt.Comm

The simplest form of `alt` statement is the a *communication* `alt`. The cases of a communication `alt`'s arms are send, receive and flush statements. See Section 4.6.1 [Ref.Task.Comm], page 36.

To execute a communication `alt`, the runtime checks all of the ports and channels involved in the arms of the statement to see if any `case` can execute without blocking. If no `case` can execute, the task blocks, and the runtime unblocks the task when a `case` *can* execute. The runtime then makes a pseudo-random choice from among the set of `case` statements that can execute, executes the statement of the `case` and branches to the block of that arm.

An example of a communication `alt` statement:

```
let chan c[int] = foo();
let port p[str] = bar();
let int x = 0;
let vec[str] strs;

alt {
    case (str s <- p) {
        vec.append(strs, s);
    }
    case (c <| x) {
        x++;
    }
}
```

4.10.22.2 Ref.Stmt.Alt.Pat

A pattern `alt` statement branches on a *pattern*. The exact form of matching that occurs depends on the pattern. Patterns consist of some combination of literals, tag constructors, variable binding specifications and placeholders (`_`). A pattern `alt` has a parenthesized *head expression*, which is the value to compare to the patterns. The type of the patterns must equal the type of the head expression.

To execute a pattern `alt` statement, first the head expression is evaluated, then its value is sequentially compared to the patterns in the arms until a match is found. The first arm with a matching `case` pattern is chosen as the branch target of the `alt`, any variables bound by the pattern are assigned to local *auto* slots in the arm's block, and control enters the block.

An example of a pattern `alt` statement:

```
type list[X] = tag(nil, cons(X, @list[X]));

let list[int] x = cons(10, cons(11, nil));

alt (x) {
    case (cons(a, cons(b, _))) {
        process_pair(a,b);
    }
    case (cons(v=10, _)) {
        process_ten(v);
    }
    case (_) {
        fail;
    }
}
```

4.10.22.3 Ref.Stmt.Alt.Type

An `alt type` statement is similar to a pattern `alt`, but branches on the *type* of its head expression, rather than the value. The head expression of an `alt type` statement must be of type `any`, and the arms of the statement are slot patterns rather than value patterns. Control branches to the arm with a `case` that matches the *actual type* of the value in the `any`.

An example of an `alt type` statement:

```
let any x = foo();

alt type (x) {
    case (int i) {
        ret i;
    }
    case (list[int] li) {
        ret int_list_sum(li);
    }
    case (list[X] lx) {
        ret list_len(lx);
    }
    case (_) {
        ret 0;
    }
}
```

4.10.23 Ref.Stmt.Prove

A `prove` statement has no run-time effect. Its purpose is to statically check (and document) that its argument constraint holds at its statement entry point. If its argument typestate does not hold, under the typestate algorithm, the program containing it will fail to compile.

4.10.24 Ref.Stmt.Check

A `check` statement connects dynamic assertions made at run-time to the static typestate system. A `check` statement takes a constraint to check at run-time. If the constraint holds at run-time, control passes through the `check` and on to the next statement in the enclosing block. If the condition fails to hold at run-time, the `check` statement behaves as a `fail` statement.

The typestate algorithm is built around `check` statements, and in particular the fact that control *will not pass* a check statement with a condition that fails to hold. The typestate algorithm can therefore assume that the (static) postcondition of a `check` statement includes the checked constraint itself. From there, the typestate algorithm can perform dataflow calculations on subsequent statements, propagating conditions forward and statically comparing implied states and their specifications. See Section 4.10.1 [Ref.Stmt.Stat], page 70.

```
fn even(&int x) -> bool {
    ret x & 1 == 0;
}

fn print_even(int x) : even(x) {
    print(x);
}

fn test() {
    let int y = 8;

    // Cannot call print_even(y) here.

    check even(y);

    // Can call print_even(y) here, since even(y) now holds.
    print_even(y);
}
```

4.10.25 Ref.Stmt.IfCheck

An `if check` statement combines a `if` statement and a `check` statement in an indivisible unit that can be used to build more complex conditional control flow than the `check` statement affords.

In fact, `if check` is a "more primitive" statement `check`; instances of the latter can be rewritten as instances of the former. The following two examples are equivalent:

Example using `check`:

```
check even(x);
print_even(x);
```

Equivalent example using `if check`:

```
if check even(x) {
    print_even(x);
} else {
    fail;
}
```

4.11 Ref.Run

The Rust *runtime* is a relatively compact collection of C and Rust code that provides fundamental services and datatypes to all Rust tasks at run-time. It is smaller and simpler than many modern language runtimes. It is tightly integrated into the language's execution model of slots, tasks, communication, reflection, logging and signal handling.

4.11.1 Ref.Run.Mem

The runtime memory-management system is based on a *service-provider interface*, through which the runtime requests blocks of memory from its environment and releases them back to its environment when they are no longer in use. The default implementation of the service-provider interface consists of the C runtime functions `malloc` and `free`.

The runtime memory-management system in turn supplies Rust tasks with facilities for allocating, extending and releasing stacks, as well as allocating and freeing exterior values.

4.11.2 Ref.Run.Type

The runtime provides C and Rust code to manage several built-in types:

- `vec`, the type of vectors.
- `str`, the type of UTF-8 strings.
- `big`, the type of arbitrary-precision integers.
- `chan`, the type of communication channels.
- `port`, the type of communication ports.
- `task`, the type of tasks.

Support for other built-in types such as simple types, tuples, records, and tags is open-coded by the Rust compiler.

4.11.3 Ref.Run.Comm

The runtime provides code to manage inter-task communication. This includes the system of task-lifecycle state transitions depending on the contents of queues, as well as code to copy values between queues and their recipients and to serialize values for transmission over operating-system inter-process communication facilities.

4.11.4 Ref.Run.Refl

The runtime reflection system is driven by the DWARF tables emitted into a crate at compile-time. Reflecting on a slot or item allocates a Rust data structure corresponding to the DWARF DIE for that slot or item.

4.11.5 Ref.Run.Log

The runtime contains a system for directing logging statements to a logging console and/or internal logging buffers. See Section 4.10.14 [Ref.Stmt.Log], page 91. Logging statements can be enabled or disabled via a two-dimensional filtering process:

- By Item

  Each *item* (module, function, iterator, object, type) in Rust has a static name-path within its crate module, and can have logging enabled or disabled on a name-path-prefix basis.

- By Task

  Each *task* in a running Rust program has a unique ownership-path through the task ownership tree, and can have logging enabled or disabled on an ownership-path-prefix basis.

Logging is integrated into the language for efficiency reasons, as well as the need to filter logs based on these two built-in dimensions.

4.11.6 Ref.Run.Sig

The runtime signal-handling system is driven by a signal-dispatch table and a signal queue associated with each task. Sending a signal to a task inserts the signal into the task's signal queue and marks the task as having a pending signal. At the next scheduling opportunity, the runtime processes signals in the task's queue using its dispatch table. The signal queue memory is charged to the task's domain; if the queue grows too big, the task will fail.

5 Index

(Index is nonexistent)