Rust

A safe, concurrent, practical language.

Graydon Hoare
Mozilla Foundation

http://rust-lang.org
Version: prerelease (2b85817 2011-10-27 16:27:47 -0700)

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 expression 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 values

  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 manipulate one another's local memory. Breaking the rule of task isolation is possible only by calling external (C/C++) code.

  Inter-task communication is typed, asynchronous, and simplex, based on passing messages over channels to ports.

- 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 debug information (DWARF) and are compatible with existing, standard low-level tools (disassemblers, debuggers, profilers, dynamic loaders). The compiled units include custom metadata that carries full type and version information.

  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.

- Static control over memory allocation, packing and aliasing.

  Many values in Rust are allocated *within* their containing stack-frame or parent structure. Numbers, records, tuples and tags are all allocated this way. To allocate such values in the heap, they must be explicitly *boxed*. A *box* is a pointer to a heap allocation that holds another value, its *content*. Boxes may be either shared or unique, depending on which sort of storage management is desired.

  Boxing and unboxing in Rust is explicit, though in some cases (such as name-component dereferencing) Rust will automatically dereference a box to access its content. Box values can be passed and assigned independently, like pointers in C; the difference is that in Rust they always point to live contents, and are not subject to pointer arithmetic.

  In addition to boxes, Rust supports a kind of pass-by-pointer slot called a reference. Forming or releasing a reference does not perform reference-count operations; references can only be formed on values that will provably outlive the reference. References are not "general values", in the sense that they cannot be independently manipulated. They are a lot like C++'s references, except that they are safe: the compiler ensures that they always point to live values.

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

- Immutable data by default

  All types in Rust are immutable by default. A field within a type must be declared as `mutable` in order to be modified.

- Move semantics and unique pointers

  Rust differentiates copying values from moving them, and permits moving and swapping values explicitly rather than copying. Moving can be more efficient and, crucially, represents an indivisible transfer of ownership of a value from its source to its destination.

  In addition, pointer types in Rust come in several varieties. One important type of pointer related to move semantics is the *unique* pointer, denoted `~`, which is statically guaranteed to be the only pointer pointing to its referent at any given time.

  Combining move-semantics and unique pointers, Rust permits a very lightweight form of inter-task communication: values are sent between tasks by moving, and only types composed of unique pointers can be sent. This statically ensures there can never be sharing of data between tasks, while keeping the costs of transferring data between tasks as cheap as moving a pointer.

- 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 is an "unsafe" action, and can only be taken within a block marked with the `unsafe` keyword. Every unsafe block in a Rust compilation unit must be explicitly authorized in the crate file.

- Structural algebraic data types

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

- 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 without a type, 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.

- 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 evaluates the special `fail` expression, 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.

- Supervision hierarchy

  Rust has a system for propagating task-failures, either directly to a supervisor task, or indirectly by sending a message into a channel.

- Resource types with deterministic destruction

  Rust includes a type constructor for *resource* types, which have an associated destructor and cannot be moved in memory. Resources types belong to the kind of *pinned* types, and any value that directly contains a resource is implicitly pinned as well.

  Resources can only contain types from the pinned or unique kinds of type, which means that unlike finalizers, there is always a deterministic, top-down order to run the destructors of a resource and its sub-resources.

- Typestate system

  Every storage slot in a Rust frame 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 within each frame. The static calculation of the typestates of a program is a function-local 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 values, typestates statically model *assertions that hold* before and after statements and expressions.

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 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 whitespace, keywords, operators and structural symbols) are drawn from the ASCII-compatible range of Unicode. Identifiers are drawn from Unicode characters specified by the `XID_start` and `XID_continue` rules given by UAX #31[1]. String and character literals may include the full range of Unicode characters.

TODO: formalize this section much more.

4.1.1 Ref.Lex.Ignore

Characters considered to be *whitespace* or *comment* are ignored, and are not considered as tokens. They serve only to delimit tokens. Rust is otherwise a free-form language.

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 *single-line comments* or *multi-line comments*.

A *single-line comment* is 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.

A *multi-line comments* is any sequence of Unicode characters beginning with U+002F U+002A (`"/*"`) and ending with U+002A U+002F (`"*/"`), *excluding* cases in which such a sequence occurs within a string literal token. Multi-line comments may be nested.

4.1.2 Ref.Lex.Ident

Identifiers follow the rules given by Unicode Standard Annex #31, in the form closed under NFKC normalization, *excluding* those tokens that are otherwise defined as keywords or reserved tokens. See Section 4.1.3 [Ref.Lex.Key], page 14. See Section 4.1.4 [Ref.Lex.Res], page 14.

That is: an identifier starts with any character having derived property `XID_Start` and continues with zero or more characters having derived property `XID_Continue`; and such an identifier is NFKC-normalized during lexing, such that all subsequent comparison of identifiers is performed on the NFKC-normalized forms.

TODO: define relationship between Unicode and Rust versions.

[2]

[1] Unicode Standard Annex #31: Unicode Identifier and Pattern Syntax

[2] This identifier syntax is a superset of the identifier syntaxes of C and Java, and is modeled on Python PEP #3131, which formed the definition of identifiers in Python 3.0 and later.

4.1.3 Ref.Lex.Key

The keywords are:

```
use          syntax     mutable    native     unchecked
mod          import     export     let        const
auth         unsafe     as         self       log
bind         type       true       false      any
int          uint       float      char       bool
u8           u16        u32        u64        f32
i8           i16        i32        i64        f64
tag          vec        str        with       fn
iter         pure       obj        resource   if
else         alt        in         do         while
break        cont       note       assert     claim
check        prove      fail       for        each
ret          put        be
```

4.1.4 Ref.Lex.Res

The reserved tokens are:

```
f16          f80        f128
m32          m64        m128       dec
```

At present these tokens have no defined meaning in the Rust language.

These tokens may correspond, in some current or future implementation, to additional built-in types for decimal floating-point, extended binary and interchange floating-point formats, as defined in the IEEE 754-1985 and IEEE 754-2008 specifications.

4.1.5 Ref.Lex.Num

A *number literal* is either an *integer literal* or a *floating-point literal*.

An *integer literal* has one of three forms:

1. A *decimal literal* starts with a *decimal digit* and continues with any mixture of *decimal digits* and *underscores*.
2. A *hex literal* starts with the character sequence U+0030 U+0078 (`"0x"`) and continues as any mixture *hex digits* and *underscores*.
3. A *binary literal* starts with the character sequence U+0030 U+0062 (`"0b"`) and continues as any mixture *binary digits* and *underscores*.

By default, an integer literal is of type `int`. An integer literal may be followed (immediately, without any spaces) by a *integer suffix*, which changes the type of the literal. There are three kinds of integer literal suffix:

1. The `u` suffix gives the literal type `uint`.
2. The `g` suffix gives the literal type `big`.
3. Each of the signed and unsigned machine types `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64` and `i64` give the literal the corresponding machine type.

A *floating-point literal* has one of two forms:

1. Two *decimal literals* separated by a period character U+002E ('.'), with an optional *exponent* trailing after the second *decimal literal*.
2. A single *decimal literal* followed by an *exponent*.

By default, a floating-point literal is of type `float`. A floating-point literal may be followed (immediately, without any spaces) by a *floating-point suffix*, which changes the type of the literal. There are only two floating-point suffixes: `f32` and `f64`. Each of these gives the floating point literal the associated type, rather than `float`.

A set of suffixes are also reserved to accommodate literal support for types corresponding to reserved tokens. The reserved suffixes are `f16`, `f80`, `f128`, `m`, `m32`, `m64` and `m128`.

A *hex digit* is either a *decimal digit* or else a character in the ranges U+0061-U+0066 and U+0041-U+0046 (`'a'-'f'`, `'A'-'F'`).

A *binary digit* is either the character U+0030 or U+0031 (`'0'` or `'1'`).

An *exponent* begins with either of the characters U+0065 or U+0045 (`'e'` or `'E'`), followed by an optional *sign character*, followed by a trailing *decimal literal*.

A *sign character* is either U+002B or U+002D (`'+'` or `'-'`).

Examples of integer literals of various forms:

```
123;                                // type int
123u;                               // type uint
123_u;                              // type uint
0xff00;                             // type int
0xffu8;                             // type u8
0b1111_1111_1001_0000_i32;          // type i32
0xffff_ffff_ffff_ffff_ffff_ffffg;  // type big
```

Examples of floating-point literals of various forms:

```
123.0;                              // type float
0.1;                                // type float
0.1f32;                             // type f32
12E+99_f64;                         // type f64
```

4.1.6 Ref.Lex.Text

A *character literal* is a single Unicode character enclosed within two U+0027 (single-quote) characters, with the exception of U+0027 itself, which must be *escaped* by a preceding U+005C character ('\').

A *string literal* is a sequence of any Unicode characters enclosed within two U+0022 (double-quote) characters, with the exception of U+0022 itself, which must be *escaped* by a preceding U+005C character ('\').

Some additional *escapes* are available in either character or string literals. An escape starts with a U+005C ('\') and continues with one of the following forms:

- An *8-bit codepoint escape* escape starts with U+0078 ('x') and is followed by exactly two *hex digits*. It denotes the Unicode codepoint equal to the provided hex value.
- A *16-bit codepoint escape* starts with U+0075 ('u') and is followed by exactly four *hex digits*. It denotes the Unicode codepoint equal to the provided hex value.
- A *32-bit codepoint escape* starts with U+0055 ('U') and is followed by exactly eight *hex digits*. It denotes the Unicode codepoint equal to the provided hex value.
- A *whitespace escape* is one of the characters U+006E, U+0072, or U+0074, denoting the unicode values U+000A (LF), U+000D (CR) or U+0009 (HT) respectively.
- The *backslash escape* is the character U+005C ('\') which must be escaped in order to denote *itself*.

4.1.7 Ref.Lex.Syntax

Syntactic extensions are marked with the *pound* sigil U+0023 (`#`), followed by an identifier, one of `fmt`, `env`, `concat_idents`, `ident_to_str`, `log_syntax`, `macro`, or the name of a user-defined macro. This is followed by a vector literal. (Its value will be interpreted syntactically; in particular, it need not be well-typed.)

TODO: formalize those terms more.

4.1.8 Ref.Lex.Sym

The special symbols are:

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

4.2 Ref.Path

A *path* is a sequence of one or more path components separated by a namespace qualifier (`::`). If a path consists of only one component, it may refer to either an item or a slot in a local control scope. See Section 4.5.3 [Ref.Mem.Slot], page 23. See Section 4.7 [Ref.Item], page 29. If a path has multiple components, it refers to an item.

Every item has a *canonical path* within its crate, but the path naming an item is only meaningful within a given crate. There is no global namespace across crates; an item's canonical path merely identifies it within the crate. See Section 4.4.1 [Ref.Comp.Crate], page 19.

Path components are usually identifiers. See Section 4.1.2 [Ref.Lex.Ident], page 13. The last component of a path may also have trailing explicit type arguments.

Two examples of simple paths consisting of only identifier components:

```
x;
x::y::z;
```

In most contexts, the Rust grammar accepts a general *path*, but subsequent passes may restrict paths occurring in various contexts to refer to slots or items, depending on the semantics of the occurrence. In other words: in some contexts a slot is required (for example, on the left hand side of the copy operator, see Section 4.11.1 [Ref.Expr.Copy], page 48) and in other contexts an item is required (for example, as a type parameter, see Section 4.7 [Ref.Item], page 29). In no case is the grammar made ambiguous by accepting a general path and interpreting the reference in later passes. See Section 4.3 [Ref.Gram], page 18.

An example of a path with type parameters:

```
m::map<int,str>;
```

4.3 Ref.Gram

TODO: mostly 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.
- 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 17. 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 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,

[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.

in order to select an an appropriate imported crate. See Section 4.4.2 [Ref.Comp.Attr], page 20.

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 metadata, describing:

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

An example of a crate:

```
// Linkage attributes
#[ link(name = "projx"
        vers = "2.5",
        uuid = "9cccc5d5-aceb-4af5-8285-811211826b82") ];

// Additional metadata attributes
#[ desc = "Project X",
   license = "BSD" ];
   author = "Jane Doe" ];

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

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

4.4.2 Ref.Comp.Attr

Static entities in Rust – crates, modules and items – may have attributes applied to them.[2] An attribute is a general, free-form piece of metadata that is interpreted according to name, convention, and language and compiler version. Attributes may appear as any of:

- A single identifier, the attribute name
- An identifier followed by the equals sign '=' and a literal, providing a key/value pair
- An identifier followed by a parenthesized list of sub-attribute arguments

Attributes are applied to an entity by placing them within a hash-list (`#[...]`) as either a prefix to the entity or as a semicolon-delimited declaration within the entity body.

An example of attributes:

[2] Attributes in Rust are modeled on Attributes in ECMA-335, C#

```
// A function marked as a unit test
#[test]
fn test_foo() {
  ...
}

// General metadata applied to the enclosing module or crate.
#[license = "BSD"];

// A conditionally-compiled module
#[cfg(target_os="linux")]
module bar {
  ...
}
```

In future versions of Rust, user-provided extensions to the compiler will be able to interpret attributes. When this facility is provided, a distinction will be made between language-reserved and user-available attributes.

At present, only the Rust compiler interprets attributes, so all attribute names are effectively reserved. Some significant attributes include:

- The `cfg` attribute, for conditional-compilation by build-configuration
- The `link` attribute, describing linkage metadata for a crate
- The `test` attribute, for marking functions as unit tests.

Other attributes may be added or removed during development of the language.

4.4.3 Ref.Comp.Syntax

Rust provides a notation for *syntax extension*. The notation for invoking a syntax extension is a marked syntactic form that can appear as an expression in the body of a Rust program. See Section 4.1.7 [Ref.Lex.Syntax], page 16.

After parsing, a syntax-extension incovation is expanded into a Rust expression. The name of the extension determines the translation performed. In future versions of Rust, user-provided syntax extensions aside from macros will be provided via external crates.

At present, only a set of built-in syntax extensions, as well as macros introduced inline in source code using the `macro` extension, may be used. The current built-in syntax extensions are:

- `fmt` expands into code to produce a formatted string, similar to `printf` from C.
- `env` expands into a string literal containing the value of that environment variable at compile-time.
- `concat_idents` expands into an identifier which is the concatenation of its arguments.
- `ident_to_str` expands into a string literal containing the name of its argument (which must be a literal).
- `log_syntax` causes the compiler to pretty-print its arguments.

Finally, `macro` is used to define a new macro. A macro can abstract over second-class Rust concepts that are present in syntax. The arguments to `macro` are a bracketed list of pairs (two-element lists). The pairs consist of an invocation and the syntax to expand into. An example:

```
#macro[[#apply[fn, [args, ...]], fn(args, ...)]];
```

In this case, the invocation `#apply[sum, 5, 8, 6]` expands to `sum(5,8,6)`. If `...` follows an expression (which need not be as simple as a single identifier) in the input syntax, the matcher will expect an arbitrary number of occurences of the thing preceeding it, and bind syntax to the identifiers it contains. If it follows an expression in the output syntax, it will transcribe that expression repeatedly, according to the identifiers (bound to syntax) that it contains.

The behavior of `...` is known as Macro By Example. It allows you to write a macro with arbitrary repetition by specifying only one case of that repetition, and following it by `...`, both where the repeated input is matched, and where the repeated output must be transcribed. A more sophisticated example:

```
#macro[#zip_literals[[x, ...], [y, ...]],
       [[x, y], ...]];
#macro[#unzip_literals[[x, y], ...],
       [[x, ...], [y, ...]]];
```

In this case, `#zip_literals[[1,2,3], [1,2,3]]` expands to `[[1,1],[2,2],[3,3]]`, and `#unzip_literals[[1,1], [2,2], [3,3]]` expands to `[[1,2,3],[1,2,3]]`.

Macro expansion takes place outside-in: that is, `#unzip_literals[#zip_literals[[1,2,3],[1,2,3]]]` will fail because `unzip_literals` expects a list, not a macro invocation, as an argument.

The macro system currently has some limitations. It's not possible to destructure anything other than vector literals (therefore, the arguments to complicated macros will tend to be an ocean of square brackets). Macro invocations and `...` can only appear in expression positions. Finally, macro expansion is currently unhygienic. That is, name collisions between macro-generated and user-written code can cause unintentional capture.

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 consist of *slots*, and allocations in the heap consist of *boxes*.

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 boxes: shared boxes – which may be subject to garbage collection – and unique boxes. The lifetime of an allocation in the heap depends on the lifetime of the box values pointing to it. Since box values may themselves be passed in and out of frames, or stored in the heap, heap allocations may outlive the frame they are allocated within.

4.5.2 Ref.Mem.Own

A task owns all memory it can *safely* reach through local variables, shared or unique boxes, and/or references. Sharing memory between tasks can only be accomplished using *unsafe* constructs, such as raw pointer operations or calling C code.

When a task sends a value of *unique* kind over a channel, it loses ownership of the value sent and can no longer refer to it. This is statically guaranteed by the combined use of "move semantics" and unique kinds, within the communication system.

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

A shared box may (in the case of a recursive, mutable shared type) be cyclic; in this case the release of memory inside the shared structure may be deferred until task-local garbage collection can reclaim it. Code can ensure no such delayed deallocation occurs by restricting itself to unique boxes and similar unshared kinds of data.

When a task finishes, its stack is necessarily empty and it therefore has no references to any boxes; the remainder of its heap is immediately freed.

4.5.3 Ref.Mem.Slot

A task's stack contains slots.

A *slot* is a component of a stack frame. A slot is either *local* or an *alias*.

A *local* slot (or *stack-local* allocation) holds a value directly, allocated within the stack's memory. The value is a part of the stack frame.

A *reference* references a value outside the frame. It may refer to a value allocated in another frame *or* a boxed value in the heap. The reference-formation rules ensure that the referent will outlive the reference.

Local slots are always implicitly mutable.

Local slots are not initialized when allocated; the entire frame worth of local slots are allocated at once, on frame-entry, in an uninitialized state. Subsequent statements within a function may or may not initialize the local slots. Local slots can be used only after they have been initialized; this condition is guaranteed by the typestate system.

References are created for function arguments. If the compiler can not prove that the referred-to value will outlive the reference, it will try to set aside a copy of that value to refer to. If this is not sematically safe (for example, if the referred-to value contains mutable fields), it will reject the program. If the compiler deems copying the value expensive, it will warn.

A function can be declared to take an argument by mutable reference. This allows the function to write to the slot that the reference refers to.

An example function that accepts an value by mutable reference:

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

4.5.4 Ref.Mem.Box

A *box* is a reference to a heap allocation holding another value. There are two kinds of boxes: *shared boxes* and *unique boxes*.

A *shared box* type or value is constructed by the prefix *at* sigil `@`.

A *unique box* type or value is constructed by the prefix *tilde* sigil `~`.

Multiple shared box values can point to the same heap allocation; copying a shared box value makes a shallow copy of the pointer (optionally incrementing a reference count, if the shared box is implemented through reference-counting).

Unique box values exist in 1:1 correspondence with their heap allocation; copying a unique box value makes a deep copy of the heap allocation and produces a pointer to the new allocation.

An example of constructing one shared box type and value, and one unique box type and value:

```
let x: @int = @10;
let x: ~int = ~10;
```

Some operations implicitly dereference boxes. Examples of such *implicit dereference* operations are:

- arithmetic operators (`x + y - z`)
- field selection (`x.y.z`)

An example of an implicit-dereference operation performed on box values:

```
let x: @int = @10;
let y: @int = @12;
assert (x + y == 22);
```

Other operations act on box values as single-word-sized address values. For these operations, to access the value held in the box requires an explicit dereference of the box value.

Explicitly dereferencing a box is indicated with the unary *star* operator `*`. Examples of such *explicit dereference* operations are:

- copying box values (`x = y`)
- passing box values to functions (`f(x,y)`)

An example of an explicit-dereference operation performed on box values:

```
fn takes_boxed(b: @int) {
}

fn takes_unboxed(b: int) {
}

fn main() {
    let x: @int = @10;
    takes_boxed(x);
    takes_unboxed(*x);
}
```

4.6 Ref.Task

An 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* blocks, 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 is implicitly boxed and mutable; as such a port has a unique per-task identity and cannot be replicated or transmitted. If a port value is copied, both copies refer to the *same* port. 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.

Channels are weak: a channel does not keep the port it is bound to alive. Ports are owned by their allocating task and cannot be sent over channels; if a task dies its ports die with it, and all channels bound to those ports no longer function. Messages sent to a channel connected to a dead port will be dropped.

Channels are immutable types with meaning known to the runtime; channels can be sent over channels.

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. The types of messages must be of *unique* kind.

Messages are generally sent asynchronously, with optional rate-limiting on the transmit side. A channel contains a message queue and asynchronously sending a message merely inserts it into the sending channel's queue; message receipt is the responsibility of the receiving task.

Messages are sent on channels and received on ports using standard library functions.

[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 evaluates a communication expression on a port or channel that cannot be immediately completed. When the communication expression 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 evaluation of a `fail` expression. 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. Running the destructor code causes a temporary transition to a *running* state, and allows the destructor code to cause any subsequent state transitions. The original task of unwinding and failing thereby 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. Once a task has temporarily suspended its unwinding in the *failing* state, failure occurring from within this destructor results in *hard* failure. The unwinding procedure of hard failure frees resources but does not execute destructors. The original (soft) failure is still resumed at the point where it was temporarily suspended.

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.Sched

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 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.

4.6.4 Ref.Task.Spawn

A call to `std::task::spawn`, passing a 0-argument function as its single argument, causes the runtime to construct a new task executing the passed function. The passed function

is referred to as the *entry function* for the spawned task, and any captured environment is carries is moved from the spawning task to the spawned task before the spawned task begins execution.

The result of a `spawn` call is a `std::task::task` value.

An example of a `spawn` call:

```
import std::task::*;
import std::comm::*;

fn helper(c: chan<u8>) {
    // do some work.
    let result = ...;
    send(c, result);
}

let p: port<u8>;

spawn(bind helper(chan(p)));
// let task run, do other things.
// ...
let result = recv(p);
```

4.6.5 Ref.Task.Send

Sending a value into a channel is done by a library call to `std::comm::send`, which takes a channel and a value to send, and moves the value into the channel's outgoing buffer.

An example of a send:

```
import std::comm::*;
let c: chan<str> = ...;
send(c, "hello, world");
```

4.6.6 Ref.Task.Recv

Receiving a value is done by a call to the `recv` method, on an object of type `std::comm::port`. This call 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 moves a value from the head of one of the task queues to the call's return value, and un-blocks the receiving task. See Section 4.12.3 [Ref.Run.Comm], page 56.

An example of a *receive*:

```
import std::comm::*;
let p: port<str> = ...;
let s: str = recv(p);
```

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 five primary kinds of item: modules, functions, iterators, objects and type definitions.

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 18.

Functions, iterators, objects and type definitions may be *parametrized* by type. Type parameters are given as a comma-separated list of identifiers enclosed in angle 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 angle brackets. In practice, the type-inference system can usually 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 19.

An example of imports:

```
import std::math::sin;
import std::option::*;
import std::str::{char_at, hash};

fn main() {
    // Equivalent to 'log std::math::sin(1.0);'
    log sin(1.0);
    // Equivalent to 'log std::option::some(1.0);'
    log some(1.0);
    // Equivalent to 'log std::str::hash(std::str::char_at("foo"));'
    log hash(char_at("foo"));
}
```

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(x: int, y: int) {
        ...
    }
}

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

Multiple items may be exported from a single export declaration:

```
mod foo {
    export primary, secondary;

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

    fn secondary() {
        ...
    }

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

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 4.8.12 [Ref.Type.Fn], page 40.

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

An example of a function:

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

A special kind of function can be declared with a `!` character where the output slot type would normally be. For example:

```
fn my_err(s: str) -> ! {
    log s;
    fail;
}
```

We call such functions "diverging" because they never return a value to the caller. Every control path in a diverging function must end with a `fail` or a call to another diverging function on every control path. The `!` annotation does *not* denote a type. Rather, the

result type of a diverging function is a special type called ⊥ ("bottom") that unifies with any type. Rust has no syntax for ⊥.

It might be necessary to declare a diverging function because as mentioned previously, the typechecker checks that every control path in a function ends with a `ret`, `be`, or diverging expression. So, if `my_err` were declared without the `!` annotation, the following code would not typecheck:

```
fn f(i: int) -> int {
   if i == 42 {
     ret 42;
   }
   else {
     my_err("Bad number!");
   }
}
```

The typechecker would complain that `f` doesn't return a value in the `else` branch. Adding the `!` annotation on `my_err` would express that `f` requires no explicit `ret`, as if it returns control to the caller, it returns a value (true because it never returns control).

4.7.3 Ref.Item.Pred

Any pure boolean function is called a *predicate*, and may be used as part of the static typestate system. See Section 4.9.3 [Ref.Typestate.Constr], page 43. A predicate declaration is identical to a function declaration, except that it is declared with the additional keyword `pure`. In addition, the typechecker checks the body of a predicate with a restricted set of typechecking rules. A predicate

- may not contain a `put`, `send`, `recv`, assignment, or self-call expression; and
- may only call other predicates, not general functions.

An example of a predicate:

```
pure fn lt_42(x: int) -> bool {
    ret (x < 42);
}
```

A non-boolean function may also be declared with `pure fn`. This allows predicates to call non-boolean functions as long as they are pure. For example:

```
pure fn pure_length<@T>(ls: list<T>) -> uint { /* ... */ }

pure fn nonempty_list<@T>(ls: list<T>) -> bool { pure_length(ls) > 0u }
```

In this example, `nonempty_list` is a predicate—it can be used in a typestate constraint—but the auxiliary function `pure_length` is not.

ToDo: should actually define referential transparency.

The effect checking rules previously enumerated are a restricted set of typechecking rules meant to approximate the universe of observably referentially transparent Rust procedures conservatively. Sometimes, these rules are *too* restrictive. Rust allows programmers to violate these rules by writing predicates that the compiler cannot prove to be referentially transparent, using an escape-hatch feature called "unchecked blocks". When writing

code that uses unchecked blocks, programmers should always be aware that they have an obligation to show that the code *behaves* referentially transparently at all times, even if the compiler cannot *prove* automatically that the code is referentially transparent. In the presence of unchecked blocks, the compiler provides no static guarantee that the code will behave as expected at runtime. Rather, the programmer has an independent obligation to verify the semantics of the predicates they write.

ToDo: last two sentences are vague.

An example of a predicate that uses an unchecked block:

```
fn pure_foldl<@T, @U>(ls: list<T>, u: U, f: block(&T, &U) -> U) -> U {
    alt ls {
      nil. { u }
      cons(hd, tl) { f(hd, pure_foldl(*tl, f(hd, u), f)) }
    }
}

pure fn pure_length<@T>(ls: list<T>) -> uint {
    fn count<T>(_t: T, u: uint) -> uint { u + 1u }
    unchecked {
        pure_foldl(ls, 0u, count)
    }
}
```

Despite its name, `pure_foldl` is a `fn`, not a `pure fn`, because there is no way in Rust to specify that the higher-order function argument `f` is a pure function. So, to use `foldl` in a pure list length function that a predicate could then use, we must use an `unchecked` block wrapped around the call to `pure_foldl` in the definition of `pure_length`.

4.7.4 Ref.Item.Iter

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

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 point 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 evaluates a `ret` or `be` expression of type `()` and completes its execution.

An iterator can be called only in the loop header of a matching `for each` loop or as the argument in a `put each` expression. See Section 4.11.14 [Ref.Expr.Foreach], page 52.

An example of an iterator:

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

```
}

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

4.7.5 Ref.Item.Obj

An *object item* defines the *state* and *methods* of a set of *object values*. Object values have object types. See Section 4.8.14 [Ref.Type.Obj], page 40.

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(state: @mutable int) {
    fn incr() {
       *state += 1;
    }
    fn get() -> int {
       ret *state;
    }
}

let c: counter = counter(@mutable 1);

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

Inside an object's methods, you can make *self-calls* using the `self` keyword.

```
obj my_obj() {
  fn get() -> int {
    ret 3;
  }
  fn foo() -> int {
    let c = self.get();
    ret c + 2;
  }
}

let o = my_obj();
assert o.foo() == 5;
```

Rust objects are extendable with additional methods and fields using *anonymous object* expressions. See Section 4.11.22 [Ref.Expr.AnonObj], page 55.

4.7.6 Ref.Item.Type

A *type definition* defines a set of possible values in memory. See Section 4.8 [Ref.Type], page 37. Type definitions are declared with the keyword `type`. Every value has a single, specific type; the type-specified aspects of a value include:

- 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.
- The *kind* of the type (pinned, unique or shared).

For example, the type `{x: u8, y: u8}` defines the set of immutable values that are composite records, each containing two unsigned 8-bit integers accessed through the components `x` and `y`, and laid out in memory with the `x` component preceding the `y` component. This type is of *unique* kind, meaning that there is no shared substructure with other types, but it can be copied and moved freely.

4.7.7 Ref.Item.Tag

A tag item simultaneously declares a new nominal tag type (see Section 4.8.11 [Ref.Type.Tag], page 39) as well as a set of *constructors* that can be used to create or pattern-match values of the corresponding tag type.

The constructors of a `tag` type may be recursive: that is, each constructor may take an argument that refers, directly or indirectly, to the tag type the constructor is a member of. Such recursion has restrictions:

- Recursive types can be introduced only through `tag` constructors.
- A recursive `tag` item must have at least one non-recursive constructor (in order to give the recursion a basis case).
- The recursive argument of recursive tag constructors must be *box* values (in order to bound the in-memory size of the constructor).
- 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` item and its use:

```
tag animal {
  dog;
  cat;
}

let a: animal = dog;
a = cat;
```

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

```
tag list<T> {
  nil;
  cons(T, @list<T>);
}
```

```
let a: list<int> = cons(7, @cons(13, @nil));
```

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 4.8.15 [Ref.Type.Constr], page 41.

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 a boxed 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` expression, in which the reflection is used to select a block corresponding to a particular type extraction. See Section 4.11.16 [Ref.Expr.Alt], page 52.

4.8.2 Ref.Type.Mach

The machine types are the following:

- The unsigned 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 754-2008 `binary32` and `binary64` floating-point types: `f32` and `f64`, respectively.

4.8.3 Ref.Type.Int

The Rust type `uint`[1] is an 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.

4.8.4 Ref.Type.Float

The Rust type `float` is a machine-specific type equal to one of the supported Rust floating-point machine types (`f32` or `f64`). It is the largest floating-point type that is directly supported by hardware on the target machine, or if the target machine has no floating-point hardware support, the largest floating-point type supported by the software floating-point library used to support the other floating-point machine types.

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

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

Note that due to the preference for hardware-supported floating-point, the type `float` may not be equal to the largest *supported* floating-point type.

4.8.5 Ref.Type.Prim

The primitive types are the following:

- The "nil" type `()`, having the single "nil" value `()`.[3]
- The boolean type `bool` with values `true` and `false`.
- The machine types.
- The machine-dependent integer and floating-point types.

4.8.6 Ref.Type.Big

The Rust type `big`[4] 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.

4.8.7 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.

4.8.8 Ref.Type.Rec

The record type-constructor forms a new heterogeneous product of values.[5] Fields of a record type are accessed by name and are arranged in memory in the order specified by the record type.

An example of a record type and its use:

```
type point = {x: int, y: int};
let p: point = {x: 10, y: 11};
let px: int = p.x;
```

4.8.9 Ref.Type.Tup

The tuple type-constructor forms a new heterogeneous product of values similar to the record type-constructor. The differences are as follows:

[3] The "nil" value `()` is *not* a sentinel "null pointer" value for reference 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-size type.

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

[5] The record 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.

- tuple elements cannot be mutable, unlike record fields
- tuple elements are not named and can be accessed only by pattern-matching

Tuple types and values are denoted by listing the types or values of their elements, respectively, in a parenthesized, comma-separated list. Single-element tuples are not legal; all tuples have two or more values.

The members 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 = (int,str);
let p: pair = (10,"hello");
let (a, b) = p;
assert (b == "world");
```

4.8.10 Ref.Type.Vec

The vector type-constructor represents a homogeneous array of values of a given type. A vector has a fixed size. The kind of a vector type depends on the kind of its member type, as with other simple structural types.

An example of a vector type and its use:

```
let v: [int] = [7, 5, 3];
let i: int = v[2];
assert (i == 3);
```

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 vector. This behaviour supports idiomatic in-place "growth" of a mutable slot holding a vector:

```
let v: mutable [int] = [1, 2, 3];
v += [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.

4.8.11 Ref.Type.Tag

A *tag type* is a nominal, heterogeneous disjoint union type.[6] A `tag` *item* consists of a number of *constructors*, each of which is independently named and takes an optional tuple of arguments.

Tag types cannot be denoted *structurally* as types, but must be denoted by named reference to a *tag item* declaration. See Section 4.7.7 [Ref.Item.Tag], page 35.

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

4.8.12 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 4.9.3 [Ref.Typestate.Constr], page 43) and an output slot. See Section 4.7.2 [Ref.Item.Fn], page 31.

An example of a `fn` type:

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

let int x = add(5,7);

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

4.8.13 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 and an output slot. See Section 4.7.4 [Ref.Item.Iter], page 33.

An example of an `iter` type:

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

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

4.8.14 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 4.7.5 [Ref.Item.Obj], page 34) 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.

The kind of an object type serves as a restriction to the kinds of fields that may be stored in it. Unique objects, for example, can only carry unique values in their fields.

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

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

obj adder(x: @mutable int) {
    fn take(y: int) {
        *x += y;
    }
}

obj sender(c: chan<int>) {
    fn take(z: int) {
        std::comm::send(c, z);
    }
}

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

let p: port<int> = std::comm::mk_port();

let t1: taker = adder(@mutable 0);
let t2: taker = sender(p.mk_chan());

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

4.8.15 Ref.Type.Constr

A *constrained type* is a type that carries a *formal constraint* (see Section 4.9.3 [Ref.Typestate.Constr], page 43), 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 4.11.18 [Ref.Expr.Check], page 54.

An example of a constrained type with two separate instantiations:

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

let rng1: ordered_range = {low: 5, high: 7};
```

```
// implicit: 'check less_than(rng1.low, rng1.high);'

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

4.8.16 Ref.Type.Type

TODO.

4.9 Ref.Typestate

Rust programs have a static semantics that determine the types of values produced by each expression, as well as the *predicates* that hold over slots in the environment at each point in time during execution.

The latter semantics – the dataflow analysis of predicates holding over slots – is called the *typestate* system.

4.9.1 Ref.Typestate.Point

Control flows from statement to statement in a block, and through the evaluation of each expression, from one sub-expression to another. This sequential control flow is specified as a set of *points*, each of which has a set of points before and after it in the implied control flow.

For example, this code:

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

Consists of 2 statements, 3 expressions and 12 points:

- the point before the first statement
- the point before evaluating the static initializer `"hello, world"`
- the point after evaluating the static initializer `"hello, world"`
- the point after the first statement
- the point before the second statement
- the point before evaluating the function value `print`
- the point after evaluating the function value `print`
- the point before evaluating the arguments to `print`
- the point before evaluating the symbol `s`
- the point after evaluating the symbol `s`
- the point after evaluating the arguments to `print`
- the point after the second statement

Whereas this code:

```
print(x() + y());
```

Consists of 1 statement, 7 expressions and 14 points:

- the point before the statement

- the point before evaluating the function value `print`
- the point after evaluating the function value `print`
- the point before evaluating the arguments to `print`
- the point before evaluating the arguments to `+`
- the point before evaluating the function value `x`
- the point after evaluating the function value `x`
- the point before evaluating the arguments to `x`
- the point after evaluating the arguments to `x`
- the point before evaluating the function value `y`
- the point after evaluating the function value `y`
- the point before evaluating the arguments to `y`
- the point after evaluating the arguments to `y`
- the point after evaluating the arguments to `+`
- the point after evaluating the arguments to `print`

The typestate system reasons over points, rather than statements or expressions. 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 or expression 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 or expression, as each statement or expression is likely to change the contents of memory.

4.9.2 Ref.Typestate.CFG

Each *point* can be considered a vertex in a directed *graph*. Each kind of expression or statement implies a number of points *and edges* in this graph. The edges connect the points within each statement or expression, as well as between those points and those of nearby statements and expressions in the program. 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*.

4.9.3 Ref.Typestate.Constr

A *predicate* is a pure boolean function declared with the keyword `pred`. See Section 4.7.3 [Ref.Item.Pred], page 32.

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

For example, consider the following code:

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

fn test() {
```

```
    let x: int = 10;
    let y: int = 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.

4.9.4 Ref.Typestate.Cond

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

Each *point* has an associated *condition*:

- The *precondition* of a statement or expression is the condition required at in the point before it.
- The *postcondition* of a statement or expression is the condition enforced in the point after it.

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

4.9.5 Ref.Typestate.State

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

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

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 and expression in a block, and computes a condition for each typestate. Specifically:

- Initially, every typestate is empty.
- Each statement or expression's poststate is given the union of the its prestate, precondition, and postcondition.
- Each statement or expression's poststate has the difference between its precondition and postcondition removed.
- Each statement or expression's prestate is given the intersection of the poststates of every predecessor point 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.

4.9.6 Ref.Typestate.Check

The key mechanism that connects run-time semantics and compile-time analysis of typestates is the use of `check` expressions. See Section 4.11.18 [Ref.Expr.Check], page 54. A `check` expression 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 points.[7]

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 point, 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.

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

4.10 Ref.Stmt

A *statement* is a component of a block, which is in turn a component of an outer block-expression, 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.

Rust has two kinds of statement: declarations and expressions.

A declaration serves to introduce a *name* that can be used in the block *scope* enclosing the statement: all statements before and after the name, from the previous opening curly-brace (`{`) up to the next closing curly-brace (`}`).

An expression serves the dual roles of causing side effects and producing a *value*. Expressions are said to *evaluate to* a value, and the side effects are caused during *evaluation*. Many expressions contain sub-expressions as operands; the definition of each kind of expression dictates whether or not, and in which order, it will evaluate its sub-expressions, and how the expression's value derives from the value of its sub-expressions.

In this way, the structure of execution – both the overall sequence of observable side effects and the final produced value – is dictated by the structure of expressions. Blocks themselves are expressions, so the nesting sequence of block, statement, expression, and block can repeatedly nest to an arbitrary depth.

4.10.1 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.

4.10.1.1 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.

4.10.1.2 Ref.Stmt.Decl.Slot

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

- `let` *pattern optional-init*;
- `let` *pattern* : *type optional-init*;

Where *type* is a type expression, *pattern* is an irrefutable pattern (often just the name of a single slot), and *optional-init* is an optional initializer. If present, the initializer consists of either an equals sign (`=`) or move operator (`<-`), followed by an 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 former form, with no type annotation, causes the compiler to infer the static type of the slot through unification with the types of values assigned to the slot in the remaining code in the block scope. Inference only occurs on frame-local slots, not argument slots. Function, iterator and object signatures must always declared types for all argument slots. See Section 4.5.3 [Ref.Mem.Slot], page 23.

4.10.2 Ref.Stmt.Expr

An *expression statement* is one that evaluates an expression and drops its result. The purpose of an expression statement is often to cause the side effects of the expression's evaluation.

4.11 Ref.Expr

4.11.1 Ref.Expr.Copy

A *copy expression* consists of an *lval* followed by an equals-sign (`=`) and a primitive expression. See Section 4.11 [Ref.Expr], page 48.

Executing a copy expression causes the value denoted by the expression – either a value or a primitive combination of values – to be copied into the memory location denoted by the *lval*.

A copy may entail the adjustment of reference counts, execution of destructors, or similar adjustments in order to respect the path through the memory graph implied by the `lval`, as well as any existing value held in the memory being written-to. All such adjustment is automatic and implied by the `=` operator.

An example of three different copy expressions:

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

4.11.2 Ref.Expr.Call

A *call expression* invokes a function, providing a tuple of input slots and an reference slot to serve as the function's output, bound to the *lval* on the right hand side of the call. If the function eventually returns, then the expression completes.

A call expression statically requires that the precondition declared in the callee's signature is satisfied by the expression prestate. In this way, typestates propagate through function boundaries. See Section 4.9 [Ref.Typestate], page 42.

An example of a call expression:

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

4.11.3 Ref.Expr.Bind

A *bind expression* constructs a new function from an existing function.[1] The new function has zero or more of its arguments *bound* into a new, hidden boxed tuple that holds the bindings. For each concrete argument passed in the `bind` expression, 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` expression, 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` expression:

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

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

```
type single_param_fn = fn(int) -> int;

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

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

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

A `bind` expression generally stores a copy of the bound arguments in the hidden, boxed tuple, owned by the resulting first-class function. For each bound slot in the bound function's signature, space is allocated in the hidden tuple and populated with a copy of the bound value.

The `bind` expression 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` expressions.

4.11.4 Ref.Expr.Ret

Executing a `ret` expression[2] copies a value into the output slot of the current function, destroys the current function activation frame, and transfers control to the caller frame.

An example of a `ret` expression:

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

4.11.5 Ref.Expr.Put

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

A `put` expression is only valid within an iterator.[3] The current put-recipient will eventually resume the suspended iterator containing the `put` expression, either continuing execution after the `put` expression, or terminating its execution and destroying the iterator frame.

4.11.6 Ref.Expr.As

Executing an `as` expression casts the value on the left-hand side to the type on the right-hand side.

[2] A `ret` expression is analogous to a `return` expression in the C family.

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

A numeric value can be cast to any numeric type. A native pointer value can be cast to or from any integral type or native pointer type. Any other cast is unsupported and will fail to compile.

An example of an `as` expression:

```
fn avg(v: [float]) -> float {
  let sum: float = sum(v);
  let sz: float = std::vec::len(v) as float;
  ret sum / sz;
}
```

4.11.7 Ref.Expr.Fail

Executing a `fail` expression 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.

4.11.8 Ref.Expr.Log

Executing a `log` expression 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 expressions are enabled or disabled dynamically at run-time on a per-task and per-item basis. See Section 4.12.4 [Ref.Run.Log], page 56.

4.11.9 Ref.Expr.Note

A `note` expression has no effect during normal execution. The purpose of a `note` expression is to provide additional diagnostic information to the logging subsystem during task failure. See Section 4.11.8 [Ref.Expr.Log], page 50. Using `note` expressions, 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` expressions in the block sequentially, from the first expression of the block to the last. During processing, a `note` expression has equivalent meaning to a `log` expression: it causes the runtime to append the argument of the `note` to the internal logging diagnostic buffer.

An example of a `note` expression:

```
fn read_file_lines(path: str) -> [str] {
    note path;
    let r: [str];
    let f: file = open_read(path);
    for each s: str in 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 value that is marked by a `note` expression is *not* copied aside when control passes through the `note`. In other words, if a `note` expression notes a particular *lval*, and code after the `note` mutates that slot, and then a subsequent failure occurs, the *mutated* value will be logged during unwinding, *not* the original value that was denoted by the *lval* at the moment control passed through the `note` expression.

4.11.10 Ref.Expr.While

A `while` expression 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` expression 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` expression:

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

An example of a `do-while` expression:

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

4.11.11 Ref.Expr.Break

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

4.11.12 Ref.Expr.Cont

Executing a `cont` expression 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-element increment controlling the loop.

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

4.11.13 Ref.Expr.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.

Example a for loop:

```
let v: [foo] = [a, b, c];

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

4.11.14 Ref.Expr.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 txt: str;
let lines: [str];
for each s: str in str::split(txt, "\n") {
    vec::push(lines, s);
}
```

4.11.15 Ref.Expr.If

An `if` expression is a conditional branch in program control. The form of an `if` expression is a condition expression, followed by a consequent block, any number of `else if` conditions and blocks, and an optional trailing `else` block. The condition expressions must have type `bool`. If a condition expression evaluates to `true`, the consequent block is executed and any subsequent `else if` or `else` block is skipped. If a condition expression evaluates to `false`, the consequent block is skipped and any subsequent `else if` condition is evaluated. If all `if` and `else if` conditions evaluate to `false` then any `else` block is executed.

4.11.16 Ref.Expr.Alt

An `alt` expression is a multi-directional branch in program control. There are two kinds of `alt` expression: pattern `alt` expressions and `alt type` expressions.

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 expressions 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 expression following the `alt` when the case block completes.

4.11.16.1 Ref.Expr.Alt.Pat

A pattern `alt` expression 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 *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` expression, 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 pattern is chosen as the branch target of the `alt`, any variables bound by the pattern are assigned to local slots in the arm's block, and control enters the block.

An example of a pattern `alt` expression:

```
tag list<X> { nil; cons(X, @list<X>); }

let x: list<int> = cons(10, @cons(11, @nil));

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

4.11.16.2 Ref.Expr.Alt.Type

An `alt type` expression 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` expression must be of type `any`, and the arms of the expression 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` expression:

```
let x: any = 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;
```

```
    }
}
```

4.11.17 Ref.Expr.Prove

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

4.11.18 Ref.Expr.Check

A `check` expression connects dynamic assertions made at run-time to the static typestate system. A `check` expression 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 expression in the enclosing block. If the condition fails to hold at run-time, the `check` expression behaves as a `fail` expression.

The typestate algorithm is built around `check` expressions, and in particular the fact that control *will not pass* a check expression with a condition that fails to hold. The typestate algorithm can therefore assume that the (static) postcondition of a `check` expression includes the checked constraint itself. From there, the typestate algorithm can perform dataflow calculations on subsequent expressions, propagating conditions forward and statically comparing implied states and their specifications. See Section 4.9 [Ref.Typestate], page 42.

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

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

fn test() {
    let y: int = 8;

    // Cannot call print_even(y) here.

    check even(y);

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

4.11.19 Ref.Expr.Claim

A `claim` expression is an unsafe variant on a `check` expression that is not actually checked at runtime. Thus, using a `claim` implies a proof obligation to ensure—without compiler assistance—that an assertion always holds.

Setting a runtime flag can turn all `claim` expressions into `check` expressions in a compiled Rust program, but the default is to not check the assertion contained in a `claim`. The idea behind `claim` is that performance profiling might identify a few bottlenecks in the code where actually checking a given callee's predicate is too expensive; `claim` allows the code to typecheck without removing the predicate check at every other call site.

4.11.20 Ref.Expr.IfCheck

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

In fact, `if check` is a "more primitive" expression than `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;
}
```

4.11.21 Ref.Expr.Assert

An `assert` expression is similar to a `check` expression, except the condition may be any boolean-typed expression, and the compiler makes no use of the knowledge that the condition holds if the program continues to execute after the `assert`.

4.11.22 Ref.Expr.AnonObj

An *anonymous object* expression extends an existing object with methods.

4.12 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 memory, tasks, communication, reflection, logging and signal handling.

4.12.1 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 boxed values.

4.12.2 Ref.Run.Type

The runtime provides C and Rust code to assist with various built-in types, such as vectors, strings, bignums, and the low level communication system (ports, channels, tasks).

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

4.12.3 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.

4.12.4 Ref.Run.Log

The runtime contains a system for directing logging expressions to a logging console and/or internal logging buffers. See Section 4.11.8 [Ref.Expr.Log], page 50. Logging expressions 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 path within its crate module, and can have logging enabled or disabled on a path-prefix basis.

- By Task

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

4.12.5 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; if the queue grows too big, the task will fail.

5 Index

N

O

P

R

S

T

U

V

W

Y