The μ_ Specification — `r0.1`

Preface

mu_ (stylized μ_) is an extremely simple programming language. This document contains a complete specification of the language followed by a series of appendicies elucidating motivation, design decisions, implementation strategies, and so on. These appendices are not part of the formal language specification and in the event of a conflict between the specification and the content of an appendix the specification is authoritative.

This is revision `r0.1` of the specification.

1 Source

1.1 Encoding

A mu_ program (recommended extensions `.mu_`, `.muu`) consists of a series of bytes. The behaviors of bytes with the high bit set is implementation defined. As a consequence a conforming implementation MAY choose to interpret mu_ code as text in ASCII, UTF-8, or another similar encoding. This document will adopt such an interpretation when displaying mu_ source.

1.2 Syntax

1.2.1 Grammar

```
<program>  ::= <ws> <expr> <ws>

<expr>     ::= <atom> | <list>

<atom>     ::= <ordinary> { <ordinary> }

<list>     ::= <lparen>
               { <ws> <expr> }
               [ <ws> <dot> <ws> <expr> ] <ws>
               <rparen>

<ws>       ::= { <tab> | <lf> | <cr> | <sp> }

<lparen>   ::= 0×28 ;; LEFT PARENTHESIS  '('
<rparen>   ::= 0×29 ;; RIGHT PARENTHESIS ')'
<dot>      ::= 0×2e ;; FULL STOP         '.'

<tab>      ::= 0×09 ;; HORIZONTAL TAB    '\t'
<lf>       ::= 0×0a ;; LINE FEED         '\n'
<cr>       ::= 0×0d ;; CARRIAGE RETURN   '\r'
<sp>       ::= 0×20 ;; SPACE             ' '

<ordinary> ::= <any> - <special>

<any>      ::= 0×20-0×7e ;; ALL PRINTABLE ASCII
<special>  ::= <lparen> | <rparen> | <dot>
             | <tab> | <lf> | <cr> | <sp>
```

1.2.2 Interpretation

Each distinct <**atom**> in the source is an *atom name*, a sequence of bytes, and is given a distinct positive *atom number*.[1][2] How different byte sequences are mapped to numbers is completely implementation defined with the only restrictions being that:

- Atom numbers are between 1 and $2^{31} - 1$.
- Two atoms with the same name within the same execution of a program get the same atom number.
- Two atoms with different names within the same execution of a program get different atom numbers.

For example, all instances of the atom with name `hello` (hex `68 65 6c 6c 6f`) are to be assigned the same atom number (say, 3) which must be different from the atom number for `world` (hex `77 6f 72 6c 64`) (say, 7) but what exactly the atom numbers are can vary between implementations, programs, or even different executions of the same program at the implementer's discretion.

An implementation MAY assume that there are no more than $2^{11} - 1$ distinct atom names referred to in a given source text, that each atom name has a length of at most $2^7 - 1$ bytes, and that the combined length of these names (excluding duplicates) is no more than $2^{15} - 2^{11}$ bytes.[3]

[1] See 2.1 Atoms.
[2] See TODO B Implementation Strategies.
[3] These restrictions permit an implementation to allocate its atom names into a singular region of memory of size i32::MAX separated by NULLs, for instance.

A list in the source code is constructed of cons cells[4] according to the following algorithm:

```
def parse_list(tokens):
    eat a lparen from tokens
    list = parse_list_inner(tokens)
    eat a rparen from tokens
    return list

def parse_list_inner(tokens):
    if ( the next token in tokens is dot ):
        eat a dot from tokens
        return parse an expression from tokens
    elif ( the next token in tokens is rparen ):
        return the 0 atom
    else:
        head = parse an expression from tokens
        tail = parse_list_inner(tokens)
        return a new cons cell of head and tail
```

A few things to note about this algorithm:

- `()` becomes the 0 atom. For this reason the 0 atom is also called nil and is usually denoted `()`.
- A list like $(i_1\ i_2\ i_3)$ denotes an ordinary linked list $i_1 \to i_2 \to i_3 \to$ **NIL**.
- A dotted pair like $(l\ .\ r)$ denotes a simple cons cell (l, r).
- An ordinary list like $(i_1\ i_2\ i_3\ \cdots\ i_n)$ is equivalent to the dotted $(i_1\ .\ (i_2\ .\ (i_3\ .\ (\cdots\ (i_n\ .\ ())))))$.

2 Types

μ_ is dynamically typed with exactly two types: atoms and cons cells.

2.1 Atoms

An atom is fundamentally an *unsigned* 31 *bit integer*.[5]

The 0 atom represents the empty list in list-related contexts.

Some positive atoms correspond to particular byte sequences (atom names) in the source code.[6] Not all atom values have a corresponding name nor is there a standard mechanism for converting between atoms and names during the execution of a program. This correspondence is simply a mechanism to allow source code to provide human-readable names to arbitrary symbols rather than a property of the language's runtime.

[4] See 2.2 Cons Cells.
[5] See B Implementation Strategies for a rational for this unusual choice.
[6] See 1.2.2 Interpretation.

2.2 Cons Cells

A cons cell is simply an ordered pair of items (each of which may be either an atom or another cons cell).

The first item in the pair is called the *head* and the second is called the *tail*.

A list refers to a particular structure formed of either the 0 atom or a cons cell whose tail is itself a list.[7]

3 Environments

At the core of μ_'s scoping rules is the environment, a mapping from atoms to arbitrary values. This mapping is defined as a list of pairs of positive atoms and values. The first element in each pair is a variable to bind and the second is an value to bind to that variable.

For example:

```
( (hello . hello)
  (somelist . (a b c))
  (a_variable . a_value) )
```

is an environment mapping the atom `hello` to itself, the atom `somelist` to the list `(a b c)`, and the atom `a_variable` to the atom `a_value`.

An environment may contain multiple bindings for the same atom in which case the earlier binding *shadows* the later one, effectively overriding it[8].

`()` is a legal environment containing no mappings.

3.1 Lookup

Looking up the value corresponding to an atom in an environment is defined according to the following algorithm:

```
def lookup(sym, env):
    if ( env is empty ):
        return sym
    else:
        binding = the head of env
        if ( the head of binding is sym):
            return the tail of binding
        else:
            return lookup(sym, the tail of env)
```

Note that duplicate bindings are resolved in favor of the first, and missing bindings resolve to themselves.

[7] See 1.2.2 Interpretation for how these are denoted.
[8] The shadowed binding is still accessible via `~env`, though.

4 Pattern Matching

A pattern is a value which can be matched against another value (called the scrutinee) in some base environment to produce a new environment which contains everything in the base environment in addition to new bindings representing components of the value.

Every value which does not contain duplicate positive atoms is a valid pattern.

Matching a pattern against a value follows the following rules:

- `()` matches the value `()` and introduces no bindings. Attempting to match this pattern against any other object is undefined behavior.[9]
- A positive atom matches any value and introduces a binding from the pattern atom to the value.
- A cons cell matches a cons cell value, recursively pattern matching the value's head and tail against its head and tail. The order in which the bindings from the head and tail are included in the final environment is implementation defined and may even be non-deterministic. Attempting to match a cons cell pattern against any other object is undefined behavior.
- Introduced bindings shadow bindings in the base environment (i.e. are placed before) but do not remove them. This allows careful environment manipulation to recover the base environment which is necessary for writing fully hygienic macros.

4.1 Examples

- Matching the expression `()` against the pattern `()` in the base environment `((a . b))` produces the environment `((a . b))`.
- Matching the expression `(a b c d)` against the pattern `(x y . z)` in the base environment `()` produces some permutation of the environment `((x . a) (y . b) (z . (c d)))`
- Matching the expression `(a b c)` against the pattern `(x y)` is undefined behavior.
- Matching the expression `(a b)` against the pattern `(x y)` in the base environment `((x . xx))` may produce either the environment `((x . a) (y . b) (x . xx))` or `((y . b) (x . a) (x . xx))` but MUST NOT produce the environment `((x . a) (y . b))`.

[9] See 7 Undefined Behavior.

5 Evaluation

The heart of the language is evaluation. In fact, executing a program simply consists of parsing it then evaluating it in an empty environment.

An *expression* is any value to be evaluated.

Evaluation takes place in an environment e.[10]

Evaluating an expression behaves differently depending on whether the expression is the 0 atom, a positive atom, or a non-empty list.

5.1 Zero Atom

The 0 atom `()` evaluates to itself.

5.2 Positive Atoms

A positive atom a evaluates to `lookup(`a`, `e`)`.[11]

5.3 Lists

If a cons cell is evaluated it should be a valid list. An implementation MAY reject attempting to evaluate a non-list cons cell or MAY try to interpret such an expression cohesively.

A list is evaluated as a call by first evaluating its head and then, based on the result (referred to as the *receiver*), performing an operation on the remaining elements.

We will use the meta syntax of angle brackets $\langle expr \rangle_{env}$ to denote an expression which evaluates to $expr$ in the environment env and the syntax of an arrow with environment above ($\xrightarrow{env}$) to denote the evaluation of an expression to a value in an environment.

5.3.1 Builtins

If the receiver is an atom it should belong to the following list of atoms with builtin behavior. Notice that with the exception of `()` which has no corresponding name these builtin receivers all have names starting ~ (hex `7e 7e`).

`()` Called "quote", but represented by the 0 atom for technical reasons.[12] Evaluates to its first argument *unevaluated*.

$$(\langle () \rangle_e \ i) \xrightarrow{e} i$$

[10] See 3 Environments.

[11] See 3.1 Lookup for the definition of `lookup()`.

[12] Using the 0 atom as the quote builtin macro ensures that the quote macro is always accessible as the 0 atom is the only atom which cannot be rebound. This, in turn, ensures that any value is always accessible by invoking quote. The ability to access arbitrary content reliably regardless of environment is critical for hygienic macros to be possible.

~true Takes two arguments and tail evaluates[13] to its first argument. DOES NOT evaluate its second argument.

$$(\langle \sim\texttt{true}\rangle_e \ \langle t\rangle_e \ \ fe) \xrightarrow{e} t$$

~false Takes two arguments and tail evaluates to its second argument. DOES NOT evaluate its first argument.

$$(\langle \sim\texttt{false}\rangle_e \ \ te \ \langle f\rangle_e) \xrightarrow{e} f$$

~head Takes an argument and evaluates it. The result should be a cons cell. Returns the head of that cons cell.

$$(\langle \sim\texttt{head}\rangle_e \ \langle (h \ . \ t)\rangle_e) \xrightarrow{e} h$$

~tail Takes an argument and evaluates it. The result should be a cons cell. Returns the tail of that cons cell.

$$(\langle \sim\texttt{tail}\rangle_e \ \langle (h \ . \ t)\rangle_e) \xrightarrow{e} t$$

~cons Takes two arguments, evaluates them, and returns a new cons cell constructed from the first and second arguments.

$$(\langle \sim\texttt{cons}\rangle_e \ \langle h\rangle_e \ \langle t\rangle_e) \xrightarrow{e} (h \ . \ t)$$

~lte Takes two arguments and evaluates them. Returns ~true if the first is less than or equal to the second and ~false otherwise. Atoms are compared according to their numbers. Cons cells are always less than atoms. Two distinguishable cons cells do not compare equal. Two indistinguishable cons cells may or may not compare equal.

$$(\langle \sim\texttt{lte}\rangle_e \ \langle a\rangle_e \ \langle b\rangle_e) \xrightarrow{e} \begin{cases} \sim\texttt{true} & \text{if } a \leqslant b \\ \sim\texttt{false} & \text{otherwise} \end{cases}$$

~eq Takes two arguments, evaluates them, and returns ~true if the first is equal to the second and ~false otherwise. Atoms are compared according to their numbers. Cons cells are always less than atoms. Two distinguishable cons cells do not compare equal. Two indistinguishable cons cells may or may not compare equal.

$$(\langle \sim\texttt{eq}\rangle_e \ \langle a\rangle_e \ \langle b\rangle_e) \xrightarrow{e} \begin{cases} \sim\texttt{true} & \text{if } a = b \\ \sim\texttt{false} & \text{otherwise} \end{cases}$$

~add Takes two arguments and evaluates them. They should both be atoms. Returns the sum of the two atoms modulo 2^{31}.

$$(\langle \sim\texttt{add}\rangle_e \ \langle a\rangle_e \ \langle b\rangle_e) \xrightarrow{e} (a + b) \bmod 2^{31}$$

~sub Takes two arguments and evaluates them. They should both be atoms. Returns the difference of the two atoms modulo 2^{31}.

$$(\langle \sim\texttt{sub}\rangle_e \ \langle a\rangle_e \ \langle b\rangle_e) \xrightarrow{e} (a - b) \bmod 2^{31}$$

~and Takes two arguments and evaluates them. They should both be atoms. Returns the bitwise conjunction (bitwise and) of the two atoms

$$(\langle \sim\texttt{and}\rangle_e \ \langle a\rangle_e \ \langle b\rangle_e) \xrightarrow{e} (a \wedge b)$$

~or Takes two arguments and evaluates them. They should both be atoms. Returns the bitwise disjunction (bitwise or) of the two atoms

$$(\langle \sim\texttt{or}\rangle_e \ \langle a\rangle_e \ \langle b\rangle_e) \xrightarrow{e} (a \vee b)$$

~not Takes one argument and evaluates it. It should be an atom. Returns the bitwise negation (bitwise not) of the atom

$$(\langle \sim\texttt{not}\rangle_e \ \langle a\rangle_e) \xrightarrow{e} (\neg a)$$

~sl Takes two arguments and evaluates them. They should both be atoms. The second argument should be less than 31. Returns the first shifted left by the second number of bits.

$$(\langle \sim\texttt{sl}\rangle_e \ \langle a\rangle_e \ \langle b\rangle_e) \xrightarrow{e} (a << b)$$

~sr Takes two arguments and evaluates them. They should both be atoms. The second argument should be less than 31. Returns the first shifted right by the second number of bits.

$$(\langle \sim\texttt{sr}\rangle_e \ \langle a\rangle_e \ \langle b\rangle_e) \xrightarrow{e} (a >> b)$$

~env Takes no arguments. Returns the environment.

$$(\langle \sim\texttt{env}\rangle_e) \xrightarrow{e} e$$

~sys Takes two arguments. DOES NOT evaluate them. Based on the value of the first argument perform a different implementation defined operation on the second argument. See 5.3.3 The ~sys Builtin for a complete definition.

$$(\langle \sim\texttt{sys}\rangle_e \ \ sym \ \ arg) \xrightarrow{e} \text{See 5.3.3}$$

Each builtin takes a fixed number of arguments. Providing the wrong number of arguments to a builtin is undefined behavior.[14]

Attempting to use any atom that is not on this list of receivers is undefined behavior as is passing an argument of (or that evaluates to, where relevant) a type other than that expected.

[13] See 6 Required Optimizations.

[14] See 7 Undefined Behavior.

5.3.2 User Defined Receivers

If the receiver is a list then it represents a user-defined macro or function and should have exactly 2 or 3 elements, respectively.

The first of these elements is a pattern[15] and the second is the body. The third element has two purposes:

1. Its presence indicates that this receiver is a function rather than a macro.
2. It defines an environment to use when expanding the function body. Combined with the `~env` builtin, this allows lexical scoping of functions, scoped evaluation as a derived object, and more.[16]

To evaluate a call involving a user-defined receiver preform the following steps:

1. If the receiver is a function, evaluate all of its arguments, otherwise, leave them unevaluated.
2. Match the list of arguments against the included pattern to produce a new environment.[17]
    - If the receiver is a function, use the included environment as the base when constructing the new environment.
    - Otherwise, use the calling environment as the base when constructing the new environment.
3. Tail evaluate[18] the included body in the constructed environment.

Attempting to use any cons-cell which does not form a valid list of either exactly two or three elements where the first element is a valid pattern and the third (if present) is a valid environment as a receiver is undefined behavior.[19]

5.3.3 The `~sys` Builtin

`~sys` is the main point of extensibility built into μ_.

Calling the `~sys` with two 0 atoms, e.g. `((() ~sys) () ())`, should return a list of pairs mapping *system operation names* to *system operation codes*.[20] This mapping should remain constant throughout a program's lifetime.

The heads of these pairs, the *system operation names*, are atoms which correspond to human readable names of a sort a programmer may write in their source code. The tails, meanwhile, the *system operation codes*, should also be atoms but which need not be human readable names.

In order to call a system operation, the system operation code is used as the first argument to `~sys` and the desired argument is used as the second argument. The code, argument, and calling environment are then handled in some implementation-specific way. Neither the code nor argument is evaluated prior to calling, but the system operation has all the information it needs to evaluate the argument if it wishes to.

Using both a readable name and a code and delegating the mapping to the user's program this way allows for an extremely simple implementation of `~sys` using a jump table or similar structure while still allowing readable names to be used to refer to these operations.

6 Required Optimizations

Several points in 5 Evaluation specify to tail evaluate some expression. In such cases implementations MUST perform proper tail-call optimization. This means that evaluating an expression which causes a tail evaluation of another expression which itself causes the tail evaluation of another expression and so on should only take up $O(1)$ space for storage of the current evaluation state (e.g. call stack). This optimization is necessary to ensure programmers can write loops without fear of running out of space.

Implementations must not needlessly duplicate cons cells. Acquiring a cons cell from a binding multiple times should not require additional space for each instance.

Implementations must have a garbage collector. Concretely, repeatedly creating and then no longer using cons cells should not cause memory expenditure to grow without bound.

E Tests offers a list of test cases which, in addition to testing correctness of an implementation, test that these characteristics are met appropriately.

7 Undefined Behavior

Several points throughout this specification refer to certain occurrences as undefined behavior. Implementations are free to adopt arbitrary behavior in these instances including but not limited to:

- Extending the specification behavior
- Producing some kind of error
- Producing a nonsense result
- Halting and catching fire

[15] See 4 Pattern Matching.
[16] See TODO.
[17] See 4 Pattern Matching for the definition of this operation.
[18] See 6 Required Optimizations.
[19] See 7 Undefined Behavior.
[20] For a system which does not provide implementation specific functionality through this mechanism just return `()`.

Appendix A

Background

Appendix B

Implementation Strategies

Appendix C

Derivation of Higher-Level Functionality

Appendix D

Reference Implementation (WebAssembly)

Appendix E

Tests