RADICLE

v1.0

JULIAN K. ARNI

julian@monadic.xyz

JAMES HAYDON

james@monadic.xyz

Abstract. Though replicated state machines have a wide range of applications, their semantics must always possess certain properties, such as determinism. Rather than program such semantics anew, we propose a language, Radicle which substantially simplifies the process of defining such state machines. Because Radicle may be made into any RSM via *eval-redefinition*, it can be considered a universal replicated state machine.

This mechanism also allows changing the semantics of running state machines without loss of state, and with the same consensus guarantees as provided by the underlying consensus system.

1. Introduction

Replicated state machines are a widely used paradigm to program fault-tolerant systems. The paradigm involves deploying a deterministic state machine across multiple nodes (servers); these nodes can respond to client requests, and by agreeing on the order of these requests (consensus), ensure agreement on their state and output. If some fraction of nodes is unavailable (or in the looser requirement of byzantine fault tolerance, if they are responding in arbitrary ways), the overall system can still function correctly.

The service these systems replicate may be key-value stores, file-systems, append-only logs, account balances, etc. Each of these services is often re-implemented anew, leading to substantial development costs, as well as subtle bugs in the interim state of the system during the inevitable software upgrades. In this paper, we describe a language, Radicle, for defining the behaviour of replicated state machines (RSMs) independently of the underlying consensus. The language is designed so that new domain-specific languages (DSLs) can easily be defined for each service provided by the system. Each such DSL is the definition of an RSM. The expressions of the DSL are the RSMs inputs; the value such an expression evaluates to its outputs; and the changes in the environment its state changes.

Thus, if for example the RSM we are interested in is a ledger of accounts, expressions or inputs may be transfers, the state may be the balance and ownership of accounts, and outputs may be the new balances of affected accounts or an error message if the transfers are not allowed (due to insufficient funds or incorrect permissions).

Additionally, using the same mechanism as for DSL-definition (namely, an *eval redefinition*), we provide a way for upgrading the DSL itself *with the same guarantees of agreement* between nodes as the underlying consensus, reducing the coordination difficulty of an upgrade. DSLs can be defined in such a way that their *re*-definitions are themselves just one of the sorts of inputs RSM accepts; thus nodes will agree on the ordering of the upgrade with respect to other inputs, and will not go out of sync as a consequence of update.

While (crucially for the purpose of ensuring consensus) the core of the language is deterministic, Radicle also possesses an additional set of primitive operations that allow side-effects. We show how a publish-subscribe model for side-effects (or in the formalism of [6], an effect-handling central which is never provided continuations) allows responding to outputs of the state machine or state changes in a non-deterministic way, without endangering the determism of the state machine itself. This in turn makes the separation between reads and writes correct by construction.

Radicle has been developed in the context of a broader effort to create a community-owned platform for open-source development (oscoin), which includes a consensus algorithm and a networking component. The oscoin platform allows users and communities to create permissioned and permissionless RSMs with their own semantics, be it to maintain decentralized version-controled code, issues, pull requests, or collective decision-making. Radicle is oriented towards making that process as simple and clear as possible. We do not in this paper further discuss the broader oscoin system.

In the rest of the paper, we describe the language in more detail (Section 2) and show some example applications built with Radicle—first, an upgradable key-value store and then a currency (Section 3).

Date: September 2018.

2. Language Syntax and Semantics

None of the individual parts of Radicle bring anything new to programming language theory. However, as its intended domain of use differs significantly from that of other programming languages, the set of design choices that characterize it make for a unique language. Radicle strives:

(1) To be deterministic, so that Radicle programs specifiy deterministic state machines.
(2) To be powerful enough to be a 'universal state-machine'.
(3) To be able to restrict this power appropriately, so that malicious inputs can be rejected, and to aid reasoning about the behaviour of valid inputs.
(4) To be concise, expressive and emphasize correctness, so that new chains, with new semantics can be created cheaply (in terms of development time), be easily understood by all participants, and hopefully be bug-free.
(5) To work well in a collaborative setting, in which code is submitted by multiple parties in a potentially unspecified order.
(6) To have the simplest possible underlying semantics, so that Radicle interpreters are well specified, and so that Radicle can be used in security sensitive situations.
(7) To have the ability to interpret itself, so that chain semantics may be modified on-chain.

The design choices we made with Radicle are:

(a) It is a high-level, homoiconic LISP dialect.
(b) It has the ability to redefine a special function `eval`, which is used as part of its own evaluation.
(c) It has first-class functions.
(d) It is dynamically typed.
(e) It is lexically scoped with a *hyperstatic* global environment.
(f) It only has immutable values.
(g) It has a deterministic effect system for managing state (references).

	a	b	c	d	e	f	g
1							+
2	+						
3		+					
4	+		+	-		+	
5							
6	+				+	+	

Figure 1. Summary of design choices and how they interact with Radicle's design goals.

LISP is well-known for being a high-level family of languages having very concise self-interpreters, and the Scheme specification has boiled down the semantics to a small core. Choosing a LISP helps with (2), (4) and (7), and in particular basing the design on Scheme helps with (6).

The redefinition of `eval` is what enables (3).

Languages with first-class functions are particularly expressive, so (c) helps with (4).

Point (d) is a compromise: it is hard to satisfy the other points (especially (7)) with strong static types, even though this would help with (4). Though adding types to a language which can interpret itself is not impossible (see [**?**]), this constraint would also inhibit the ability to compile other languages into Radicle.

The *hyperstatic* global environment means that the resolution of free variables takes place at the definition site rather than call-site. See Section 2.3 for explanations for how this helps with (6).

The language is kept as pure and immutable as possible, which limits expressivity but emphasizes clarity and correctness (goal (4)). The abundance of mutability would also probably hinder (6).

Since chains are inherently stateful, we felt that the inclusion of a reference type for managing state would help in the specification of chain semantics. However we will discuss the alternatives.

The rest of this section goes into the details of these goals and design choices.

2.1. LISP.

Scheme has the particularity that a minimal interpreter of itself can be implemented in a few lines of code. This is achieved by:

- Code being represented by the core data-structures of the language, so that manipulating code is as straighforward as any data manipulation.
- The language being built up by a very small set of primitives: a few special forms and some primitive functions.

Let's consider a basic example of a chain which might want to change a part of its semantics. In this chain participants may `boot` certain other participants, if some condition is met, which is defined by a predicate `bootable?`. For example, `bootable?` might be defined as:

```
1  (define bootable?
2    (lambda (a b)
3      (and (admin? a) (not (admin? b)))))
```

That is, admins may boot non-admins. At some point the participants may decide that this is too crude, and that a participant should only be booted if the majority of participants agree. For the change to take place, most likely one of the inputs to the system will contain code for a new version of `bootable?` (the semantics of when such a message is to be accepted would probably be defined by yet another predicate). This code must be transmited and interpreted in the simplest and and most transparent way possible. By choosing a LISP, this code is represented by simple data-structures, and the interpretation process is very direct (simplifying parsing and abstract syntax trees, etc.). This gives the participants better confidence in understanding what the current and proposed semantics are.

2.2. Data types.

Radicle's data types are: booleans (`#t` and `#f`); strings (a sequence of characters within double quotes, with \ as the escape character), symbols, lists, dictionaries (maps), and numbers (currently only arbitrary-precision decimals). These datatypes are all immutable; additionally Radicle supports *refs*—mutable references that can hold any other datatype.

For more details on radicle values, see Appendix A.

2.3. **Hyperstatic Environments.** Most scripting languages are intended for programs where the developer can decide exactly where code should be placed, and be fully aware of the context that precedes it. One can always add a line `y = bar(x)` immediately *after* `x = foo(3)`, and not worry that perhaps something else was added in between that changes what `x` is (excepting pathological cases of distributed code collaboration).

This is not true in the environment RADICLE is intended to run. In this environment, users submit individual expressions or declarations to a running system that is *at the same time* also accepting inputs from other users. Moreover, the view of the program that the submitter has at the time of submission may *already* be outdated. This aspect is even more significant in the context of a blockchain [**?**], where expressions are queued until a block is created by a node, but what those expressions in the queue are is not accessible to other nodes.

This fact may lead to bugs or attacks. Consider the following snippet:

```
1  (define transfer (lambda (from to amount) ...))
2  (define account-alice ...)
3  (define account-bob ...)
4  (define transfer-to-alice (lambda (from amount)
5    (transfer from account-alice amount))
```

A user now wants to transfer some amount to Alice. However, in a language such as Scheme the function call `(transfer-to-alice account-bob 10)` may behave differently than this snippet alone indicates in two ways. First, `transfer-to-alice` and `account-bob`, which explicitly appear in the expression being submitted, may be shadowed by a new definition. Second, the variables `transfer`, `account-alice`, and *any other* free variables used in the body of `transfer` or transitively, in the function bodies of the functions `transfer` calls, may have been redefined.

One solution is to not allow the redefinition of variables at all. This is indeed what some smart contract [**?**] languages do, and can easily be implemented in RADICLE as well (see Section 2.5). But this comes with its own problems; one cannot define an improved version of a function that shadows the old one, and must instead accept a more and more polluted environment (which additionally may impact memory usage).

An alternative is to have a mechanism (call it `protect-at-line`) to make expressions invalid if any of the variables they rely on have been redefined since a specified line of code. However this option is quite severe. Consider:

```
1  (define x 3)
2  (define foo (lambda () x))
3  (define x 5)
```

The function call `(protect-at-line 2 (foo))` will fail despite having a perfectly reasonable interpretation—namely, `(foo)` *with* `x` *as if used after line 2* (i.e., with `x` being `3`). For more complex expressions, which may rely on many variables, this problem becomes more significant.

This reasonable interpretation is in fact precisely what hyperstatic environments provide. The function call `(foo)` in line 4 would, in this model, still evaluate to `3`. The free variables of a function refer to the values of variables *when they were defined*. This is the semantics RADICLE adopts in general (and can be found in some other languages, such as Forth and Standard ML[7]).

Now `(protect-at-line (foo))` may be given a much simpler definition: `(foo)` if `foo` *alone* has not been redefined, and otherwise an exception. If `x` has been redefined, that is no matter, since it anyhow does not change the meaning of `foo`.

2.4. `eval` **redefinition.** RADICLE should be able to emulate any state machine. Of course, most state-machines don't accept arbitrary inputs, and certainly not inputs which change the semantics of the state-machine. Thus RADICLE should have some built in mechanism for controlling how inputs get interpreted. At the start, inputs are sent directly for evaluation by the 'base' evaluator, so it is natural to associate modifying this process with redefining a function which represents evaluation. When taken to the extreme, *all* evaluation is done with a call to the special function `eval`, and this creates a reflective tower of interpreters. In RADICLE, a more limited form of `eval`-redefinition is available. As an example:

REPL
```
> (write-ref eval-ref (lambda (expr) 3))
=> ()
> 5
=> 3
```

Contrast this with the sort of `eval`-redefinition that is available in the language Black[2]:

```
1  (exec-at-metalevel
2    (let ((old-eval base-eval))
3      (set! base-eval
4            (lambda (exp env cont)
5              (write exp) (newline) (old-eval exp
                   env cont)))))
```

After this the Black REPL will behave as follows:

REPL
```
> (+ 1 2)
(+ 1 2)
+
1
2
=> 3
```

Thus, the new evaluation function is used recursively, at all levels of the expression `(+ 1 2)`.

Compare this to the following RADICLE session:

REPL
```
> (define old (read-ref eval-ref))
> (define new (lambda (e) (do (print! e) (old e))))
> (write-ref eval-ref new)
=> ()
> (+ 1 2)
(+ 1.0 2.0)
=> 3.0
```

In Black, `old-eval` calls out to `base-eval`, and in subsequent evaluation this will refer to the new definition. This is what makes the new evalution behaviour take effect at all levels of evaluation. In RADICLE, the new evaluation

only gets invoked at the topmost level, on new inputs. If one wants to make the evaluation behaviour recursive this has to be coded explicitely.

In most cases, chains will define a domain-specific language to be interpreted in some narrow way, with only some calls to the base eval to reify function definitions, in case one wants to modify the behaviour of the state-machine in some way. Furthermore, this form of evaluation redefinition doesn't (by default) create a tower of interpreters. Evaluating in a tower requires advanced interpretation methods to avoid performance overheads, and this is all the more important in Radicle's case where chains may be very long-lived. Such methods are still the subject of active reasearch (see for example [3], [9], [1], [5]), and may interfere with other requirements of the language. Moreover, since eval-redefinition affects all sub-expressions, and may affect not just the "immediate" interpreter but ones in which it is defined, reasoning about eval-redefinitions in towers can be very difficult. For these reasons we believe that this limited form of eval-redefinition suits the needs of Radicle better than the sort of reflection that can be found in languages such as Black.

For a more involved example, consider a simplistic key-value store:

```
(define store (ref dict))

(write-ref eval-ref (lambda (expr)
  (define s (read-ref store))
  (if (eq? (head expr) 'get)
      (lookup (head (tail expr)) s)
      (if (eq? (head expr) 'set)
          (modify-map
            (head (tail expr))
            (lambda (st)
                    (head (tail (tail expr))))
            store)
          'invalid-command))))
```

After which we have:

REPL

```
> (set key1 3)
=> ()
> (get key1)
=> 3
> (+ 3 2)
=> 'invalid-command
```

Note that nested expressions are not evaluated:

REPL

```
> (set key2 (+ 3 2))
=> ()
> (get key2)
=> '(+ 3 2)
```

2.5. **Connection to Reflective Towers.** The `eval`-redefinition mechanism resembles prior work on *reflective towers*. A reflective tower is an infinite sequence of interpreters (called 'levels') L_0, L_1, ..., where level L_n is interpreted by L_{n+1}. Reflective towers allow both *reification*—the ability to inspect a computation via constructs of a higher level—and *reflection*, that is, the ability to define and enter new, lower levels. Conceptually, Radicle differs from reflective towers by only allowing reflection. Thus, the only levels that exist are the ones programs create. Queinnec has quipped that reflective languages "plunge us into a world with few laws, and hardly any gravity" [8]. The more disciplined approach to reflection that Radicle takes does not possess the semantic fragility of modifying meta-meta-intepreters.

The techniques that have been developed for reducing the interpretive overhead of such towers of interpreters, however, still apply (see for example [1] and [3]), as do simpler partial-evaluation-based approaches such as [5].

2.6. **Immutable values and refs.** In Radicle all values are immutable: once created they cannot be modified in any way. This follows in the footsteps of functional programming languages such as Haskell, Clojure and Erlang.

This allows a more *local* reasoning of the behaviour of programs; values are guaranteed to not change as the result of calling a function. This is particularly important in the context of a massively collaborative program where programmers cannot rely on a global picture of the program, maintained by knowledge-dissemination via code reviews and standardized practices.

However state-machines are inherently stateful, so a reference type is included in the language. Reading or modifying the reference must be done explicitly, with the primops **`read-ref`** and **`write-ref`**, respectively.

2.7. **Effects.** In order to maintain determinism and safety in on-chain computations, the primitives available on-chain, ρ, are pure. Radicle also comes with an additional set of primitives, $\rho_!$, intended for off-chain computations, such as scripting interactions with RSMs (these primitives are by convention textually distinguished by identifiers ending with an exclamation mark).

Expressions in the RSM therefore cannot have side-effects; instead, they may evaluate to a value that *describes* an effect, but the decision of whether or how to actually carry on that effect happens at an effect-handling layer. This architecture, with a central authority administering effects received via messages, resembles work following [6] and [4].

The value that results from evaluating an expression purely may, in addition to *describing* an effect, pass a continuation with the result of that effect (if any). The effect-handling layer can chose to only call continuations without arguments. This in turn means the continuation will not have access to the result of effectful computations, though it may (at the discretion of the effect-handling layers) have output effects.

2.8. **Semantics.** Radicle's semantics are defined in terms of an abstract state machine, that is, a set of possible states S, an initial state $s_0 \in S$, a set of possible inputs I, and a transition function $\mathcal{S}\colon S \times I \to S$. For the precise definitions of the sets in use, see Appendix A.

The particularity of Radicle is that it defines a state machine whose behaviour can change in response to some input. for this reason, Radicle's semantics is defined in two steps: First we define the *base semantics* $\mathcal{B}$, which is the semantics of the 'underlying' programming language, and then the state-machine semantics $\mathcal{S}$, which specifies the particular way in which the base semantics are invoked

on each item of the input stream. Both will be defined in terms of endomaps of the *state* S of the machine, associated with each possible input, which are just elements of V, the set of *values*. Note that technically any value is permitted as an input, but in practice when deployed on a network the inputs will be deserialised from some transmission format which will most likely exclude values such as closures.

The state is composed of the *environment* E, which associates identifiers to values, and the *memory* M which tracks the values of refs:

$$S := E \times M$$

with

$$E := \text{Ident} \to 1 + V \quad \text{and} \quad M := A \to 1 + V,$$

the (implementation specific) set Ident describes the valid identifiers. The set $A = \mathbb{N} + \{\texttt{eval-addr}\}$ contains the memory addresses used for reference cells, that is, a memory address is either a natural number or the special address `eval-addr` which is used for storing the evaluation function. The set of values is defined in Appendix A.

We shall adopt the following convention to make the definitions more readable: many of the functions have as codomain a set $1 + X$ (for some set X), with the left summand indicating an error. The unique element of 1 is denoted `error`. When defining such a function in terms of another, the errors propagate naturally, so these cases will be supressed from the definitions.

2.8.1. *Base semantics* $\mathcal{B}$. The base semantics is defined as a function

$$\mathcal{B}\colon S \times V \to 1 + S \times V.$$

- Sequences: If $p_1, \ldots, p_n \in V$, then

  $$\mathcal{B}(s, (\texttt{do}\ p_1 \ldots p_n)) := \mathcal{B}^+(s, (p_i)).$$

  See below for the definition of the sequence semantics $\mathcal{B}^+$, which defines the semantics over a finite sequence of inputs (not to be confused with the state machine semantics $\mathcal{S}$ which comes later).
- Identifiers: If $i \in \text{Ident}$, then

  $$B((e, m), i) := ((e, m), e(i)),$$

  unless $e(i) = \texttt{error}$ in which case $B((e, m), i) = \texttt{error}$.
- Conditionals: For $c, t, f \in V$,

  $$\mathcal{B}(s, (\texttt{if}\ c\ t\ f)) := \begin{cases} \mathcal{B}(s', t) & \text{if } c' \neq \#\texttt{f}, \\ \mathcal{B}(s', f) & \text{otherwise,} \end{cases}$$

  where $\mathcal{B}(s, c) = (s', c')$.
  For $n \geq 0$, $c_i, x_i \in V$, $1 \leq i \leq n$,

  $$\mathcal{B}(s, (\texttt{cond}\ c_1\ x_1 \ldots c_n\ x_n)) := \begin{cases} \texttt{error} & \text{if } n = 0, \\ \mathcal{B}(s', x_1) & \text{if } v \neq \#\texttt{f}, \\ \mathcal{B}(s', (\texttt{cond}\ c_2\ x_2 \ldots c_n\ x_n)) & \text{otherwise.} \end{cases}$$

  where $\mathcal{B}(s, c_1) = (s', v)$.
- Lambdas: If $(x_1 \ldots x_n) \in I^*$ and $(p_1 \ldots p_m) \in V^*$, then

  $$\mathcal{B}((e_{\text{def}}, m_{\text{def}}), (\texttt{lambda}\ (x_1 \ldots x_n)\ p_1 \ldots p_m)) := ((e_{\text{def}}, m_{\text{def}}), f)$$

  where f is the function:

  $$((e_{\text{call}}, m_{\text{call}}), (v_1 \ldots v_{n'})) \mapsto \begin{cases} \mathcal{B}^+((e_{\text{def}}[x_i \mapsto v_i], m_{\text{call}}), (p_i)) & \text{if } n = n', \\ \texttt{error} & \text{otherwise.} \end{cases}$$

- Definitions: When $x \in \text{Ident}$ and $p \in V$,

  $$\mathcal{B}((e, m), (\texttt{define}\ x\ p)) := ((e'[x \mapsto v], m'), ())$$

  where $\mathcal{B}((e, m), p) = ((e', m'), v)$.
- Applications: When $p \in V$ and $(q_1 \ldots q_n) \in V^*$, then

  $$\mathcal{B}(s, (p\ q_1 \ldots q_n)) := f(s_n, (v_1 \ldots v_n))$$

  where

  $$\begin{aligned} \mathcal{B}(s, p) &= (s_0, f), \\ \mathcal{B}(s_0, q_1) &= (s_1, v_1), \\ &\ldots \\ \mathcal{B}(s_{n-1}, q_n) &= (s_n, v_n) \end{aligned}$$

  and f is a function. If any of these result in an error, or f is not a function, then the whole computation results in an error.
- All other inputs yield an error.

The auxiliary sequence semantics $\mathcal{B}^+$ is defined on sequences of values V^*, as follows:

$$\mathcal{B}^+(s, (p_i)_{1 \leq i \leq n}) = \begin{cases} \texttt{error} & \text{if } n = 0, \\ \mathcal{B}(s, p_1) & \text{if } n = 1, \\ \mathcal{B}^+(s', (p_i)_{2 \leq i \leq n}) & \text{otherwise,} \end{cases}$$

where $\mathcal{B}(s', p_1) = (s', v')$.

2.8.2. *State-machine semantics.* The Radicle state machine is given by an *initial state* s_0 and a transition function

$$\mathcal{S}\colon S \times V \to S.$$

The initial state is defined to be (e_0, m_0) where e_0 is the *initial environment* (see Appendix B), and

$$m_0 := (\iota_L \circ !_A)[\texttt{eval-addr} \mapsto \mathcal{B}]$$

is the memory which associates a value to the special address `eval-addr`; a function which evaluates values according to the base evaluation.

Given a value $p \in V$ and a state $(e, m) \in S$, if $m(\texttt{eval-addr})$ is undefined, or not a function, then the machine crashes. Otherwise

$$\mathcal{S}((e, m), p) = s'$$

where $m(\texttt{eval-addr})((e, m), p) = (s', v')$. That is, evaluation proceeds as specified by the original base evaluation, or any new evaluation that has been set in the special ref with address `eval-addr`.

3. Sample Programs and Chains

In this section, we develop a better sense for the language and its applications by considering sample programs.

3.1. **Self-amending key-value store.**

In Section 2, we defined a simple key-value store language. It was not, however, an *amendable* one—once defined, it was impossible to change the semantics of the running system. For short-lived chains with a narrow purpose, this might be satisfactory. For long-lived chains, the participants ideas on the purpose of the chain may morph over time. Forking to a new chain is an option, but this is not ideal for two reasons:

- Consensus on the purpose and semantics of the new chain must be achieved "off chain".
- Participants must agree on the process and logistics of migrating to a new chain, how to decide from which block this takes place, etc.

In radicle this can all take place on-chain, by updating the eval function.

```
(define password "very secret")
(define store (ref (dict)))

;; self-amending key-val store
(define starting-eval
  (lambda (expr)
    (define command (head expr))
    (cond
     (eq? command 'get) (lookup (nth 1 expr)
          (read-ref store))
     (eq? command 'set) (modify-ref store (lambda
          (s) (insert (nth 1 expr) (nth 2 expr)
          s)))
     (and (eq? command 'update)
          (eq? (nth 1 expr)
               password))
     (write-ref eval-ref (eval (nth 2 expr)))
     :else (throw 'invalid-command "Valid
          commands are: 'get', 'set' and
          'update'."))))

(write-ref eval-ref starting-eval)
```

Note that in this case, the new evaluation function that is instantiated as a result of a an `update` command, is the result of evaluating an expression with the original `eval` function, because of the hyperstatic environments. Thus, in this case, no tower of interpreters is formed. Instead, we are swapping out one eval for another.

3.2. **Currency.**

TODO: discuss how we can insure that only bob can submit the command `(transfer "bob" x n)`.

In this section we define a currency. This example demonstrates both higher order functions and data-hiding. Indeed the `create-currency` function defines all the state needed for the operation of the currency but then only returns the relevant evaluation function, making the internals of the currency unavailable to the caller. Furthermore, by returning a potential evaluation function (rather than setting it directly), this function may be used as a sub-behaviour of a more featurefull RSM.

```
;; Helper to modify a dict at a key.
(define modify-dict
  (lambda (key fn mp)
    (insert key (fn (lookup key mp)) mp)))

;; Creates a currency and returns the relevant
    evaluation function.
(define create-currency (lambda ()

  ;; The dict of all accounts
  (define accounts (ref (dict)))

  ;; Create an account with 10 coins.
  (define new-account
    (lambda (name)
      (modify-ref accounts
        (lambda (acc)
          (insert name 10 acc)))
      :ok))

  ;; Get an account's balance.
  (define balance
    (lambda (name)
      (lookup name (read-ref accounts))))

  ;; Apply a function to an account.
  (define modify-account
    (lambda (f)
      (lambda (name amount)
        (modify-ref
          accounts
          (lambda (acc)
            (modify-dict
              name
              (lambda (x)
                (f x amount))
              acc)))
        :ok)))

  ;; Debit an account.
  (define debit (modify-account -))

  ;; Credit an account.
  (define credit (modify-account +))

  ;; Transfer money from one account to another.
  (define transfer (from to amount)
    (if (< amount (balance from))
        (do (debit  from amount)
            (credit to   amount))
      fail))

  (define currency-eval
    (lambda (expr)
      (define c (head expr))
      (cond
        (eq? c 'new-account)
          (new-account (nth 1 expr))
        (eq? c 'account-balance)
          (balance (nth 2 expr))
        (eq? c 'transfer)
```

```
61              (transfer (nth 1 expr) (nth 2 expr)
                    (nth 3 expr))
62            :else fail)))
63
64      ;; We return the evaluation function for this
            currency.
65      currency-eval))
```

After loading this code, we can imagine the following RADICLE session:

REPL

```
> (write-ref eval-ref (create-currency))
=> ()
> (new-account "alice")
=> :ok
> (new-account "bob")
=> :ok
> (transfer "alice" "bob" 3)
=> :ok
> (balance "alice")
=> 7
```

3.3. **Updatable state-machine.** Most chains will operate simple state machines with state S, inputs I and transitions governed by a function

$$t \colon S \times I \to S.$$

During the lifetime of the chain, participants may decide to modify the state-machine in some way. The target state machine they have in mind is specified by states S', inputs I' and transition function $t' \colon S' \times I' \to S'$. In order for the old-state to be preserved they must also choose a way to map the old state into the new format, with a function $f \colon S \to S'$.

In order to achieve this, we define a chain that accepts two sorts of inputs: *basic inputs*:

$$(\texttt{basic}\ s), \quad s \in S$$

and *meta-inputs*:

$$(\texttt{meta}\ s), \quad s \in S$$

which are messages the participants use to coordinate in choosing a new transition function. All basic-inputs are simply handled by t to update the current state. The meta-inputs $\bar{I}$ are handled by a function

$$m \colon \bar{S} \times S \times \bar{I} \to \bar{S} + V$$

which modifies the meta-state $\bar{S}$ of the chain, while also having read-only access to the state S. When m produces a value in the right summand, this is an expression which is evaluated to a dictionary containing f, t' and possibly m'. If only f and t' are present, the state is translated to the new format with f, and the new state machine operated by t' starts accepting new basic inputs. If m' is also provided, then the protocol for redefinition of the state-machine is also updated.

Using this higher-order function we can now demonstrate several interesting examples of chains:

- Issue chain, only person who opened issue and maintainers can close, only maintainers can update the transition function.
- Ultimate example: chain managing a repo. The set of maintainers is maintained in a config file in master branch. only maintainers can accept PRs to master. Only maintainers can vote on new transition functions. This means that the function m reads not only from the vote state $\bar{S}$ but also the basic state S (i.e. the source code under version control).

References

[1] Nada Amin and Tiark Rompf. Collapsing towers of interpreters. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–33, dec 2017.

[2] Kenichi Asai. *The Reflective Language Black*. PhD thesis, University of Tokyo, 1997.

[3] Kenichi Asai, Kenichi, Asai, and Kenichi. Compiling a reflective language using MetaOCaml. *ACM SIGPLAN Notices*, 50(3):113–122, sep 2014.

[4] Andrej Bauer and Matija Pretnar. An Effect System for Algebraic Effects and Handlers. jun 2013.

[5] Matt Brown and Jens Palsberg. Jones-optimal partial evaluation by specialization-safe normalization. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–28, dec 2017.

[6] Robert Cartwright, Robert Cartwright, and Matthias Felleisen. Extensible Denotational Language Specifications. *SYMPOSIUM ON THEORETICAL ASPECTS OF COMPUTER SOFTWARE, NUMBER 789 IN LNCS*, pages 244—-272, 1994.

[7] Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. *The Definition of Standard ML (Revised)*. MIT Press, 2nd edition, 5 1997.

[8] Christian Queinnec. *Lisp in Small Pieces*. 1994.

[9] Michael Jonathan Thyer. *Lazy Specialization*. PhD thesis, York, 1999.

Appendix A. Definition of Radicle values

The set of values manipulated by the RADICLEinterpreter is defined as a coproduct of other sets (symbols, keywords, functions etc.). The definitions rely on two other sets being defined: Ident, the set of valid identifiers, and String, the set of valid strings. These can be considered implementation dependent, but the definitions for the OSCoin network can be found here: ¡formal spec website¿.

TODO: formal spec website address.

$$
\begin{aligned}
V &:= \text{Sym} + \text{Keyword} + \text{String} + \text{Bool} + \text{Num} + \text{Func} + \text{List} + \text{Dict} + \text{Ref} \\
\overline{V} &:= \text{Sym} + \text{Keyword} + \text{String} + \text{Bool} + \text{Num} + \overline{\text{List}} + \overline{\text{Dict}} \\
\text{List} &:= V^* \\
\overline{\text{List}} &:= \overline{V}^* \\
\text{Dict} &:= \overline{V} \to 1 + V \\
\overline{\text{Dict}} &:= \overline{V} \to 1 + \overline{V} \\
\text{Func} &:= S \times V^* \to S \times V \\
\text{Sym} &:= I \\
\text{Keyword} &:= I \\
\text{Bool} &:= \{\#t, \#f\} \\
\text{Ref} &:= \mathbb{N} \\
\text{Num} &:= \mathbb{Q}
\end{aligned}
$$

The canonical injections of Sym, Keyword, String, Bool, Num, Func, List, Dict and Ref into V are denoted by atom, keyword, bool, func, lis and ref respectively. However we shall often suppress these from the notation.

Appendix B. Definition of Radicle's initial environment

The initial environment is filled with a few functions that deal with manipulating the core data-structures.

B.1. **Pure functions.** *Pure functions* are those which have no effect on the state. Thus when we say `f` is defined as the pure function f, we mean that in V it is represented by:

$$(s, (v_i)_i) \mapsto (s, f((v_i)_i)).$$

In this section we only define pure functions. Furthermore, these functions will have a fixed arity. Calling a function with an argment of a different length than that in the definition results in an error.

- `eq?` tests values for equality:

$$(v_1, v_2) \mapsto \begin{cases} \texttt{t\#} & \text{if } v_1 \sim v_2, \\ \texttt{f\#} & \text{otherwise.} \end{cases}$$

  Where the relation $\sim$ is the least equivalence relation on values V such that
  - it restricts to equality on the coproduct components Sym, Keyword, String, Bool, Num and Ref,
  - $\forall (x_1 \dots x_n), (y_1 \dots y_n) \in V^*, \text{list}((x_1 \dots x_n)) \sim \text{list}((x_1 \dots x_n)) \Leftrightarrow x_1 \sim y_1, \dots, x_n \sim y_n$
  - $\forall d_1, d_2 \in (\bar{V} \to 1 + V), \text{dict}(d_1) \sim \text{dict}(d_2) \Leftrightarrow \forall x \in \bar{V}, d_1(x) \sim d_2(x)$.
- `empty-list` returns the empty-list. `cons` adds elements to the front of lists:

$$(v, (v_1 \dots v_n)) \mapsto (v v_1 \dots v_n).$$

- `head` and `tail` deconstruct lists, they correspond to the pure functions:

$$(v_1 \dots v_n) \mapsto \begin{cases} v_1 & \text{if } n \geq 1, \\ \texttt{error} & \text{otherwise} \end{cases},$$

$$(v_1 \dots v_n) \mapsto \begin{cases} (v_2 \dots v_n) & \text{if } n \geq 1, \\ \texttt{error} & \text{otherwise.} \end{cases}$$

- `empty-dict`, `lookup` and `insert` allow the creation and querying of dicts. They correspond to the pure functions:

$$
\begin{aligned}
() &\mapsto (k \mapsto \texttt{error}), \\
(k, d) &\mapsto d(k), \\
(k, v, d) &\mapsto d[k \mapsto v].
\end{aligned}
$$

- The pure function `string-append` concatenates a list of strings (implementation specific).
- The pure functions $+, *, -, <, >$ correspond to the mathematical functions $+, \times, -, <, >$ respectively.

B.2. **Impure functions.** There are only three impure functions and they deal with reference cells.

- `ref` creates a new reference cell and returns it:

$$((e, m), v) \mapsto ((e, m[a \mapsto v]), \mathrm{ref}(a))$$

  where a is a new memory address, that is, $m(a) = \texttt{error}$. How a is generated is implementation specific.
- `read-ref!` dereferences a reference cell:

$$((e, m), r) \mapsto \begin{cases} ((e, m), m(a)) & \text{if } r = \mathrm{ref}(a) \text{ for some address } a, \\ \texttt{error} & \text{otherwise.} \end{cases}$$

- `write-ref!` writes a new value to a reference cell:

$$((e, m), r, v) \mapsto \begin{cases} ((e, m[a \mapsto v]), ()) & \text{if } r = \mathrm{ref}(a) \text{ for some address } a, \\ \texttt{error} & \text{otherwise.} \end{cases}$$

- Lastly, `eval-ref` holds the evaluation function, that is the initial environment associates the symbol `eval-ref` to the value ref(`eval-addr`), where `eval-addr` $\in A$ is the special address for the evaluation function. We will usually define:

```
(define eval
  (lambda (expr)
    ((read-ref! eval-ref) expr)))
```

  so that the function `eval` evaluates expressions with the current evaluator.

Appendix C. Derived Forms

Derived forms are defined here as sample redefinitions of `eval`.

`quote.`

`quasiquote.`

`unquote.`

`def.`

```
;; TODO: doc handling

(define eval (lambda (expr)
   (if (eq? (head expr) 'def)
       ((lambda ()
         (define pat (head (tail expr)))
         (define body (head (tail (tail expr))))
         (if (list? pat)
             (if (null? pat)
                 (error "def: bad name or pattern")
                 `(define ,(head pat) (lambda ,(tail pat) ,body)))
             `(define ,pat ,body))))
       expr))
```

Example:

```
(def (factorial n)
  (:doc-str "The factorial function")
  (if (<= n 0)
      1
      (* n (factorial (- n 1)))))

;; Which is equivalent to:

(begin
  (define factorial (lambda (n)
    (if (<= n 0)
```

```
        1
        (* n (factorial (- n 1)))))
  (add-doc-str factorial "The factorial function")
```