# Erda

Version 6.5

Tero Hasu

October 14, 2016

Erda$_{RVM}$, Erda$_{RVM}{}^{\sigma}$, Erda$_{C++}$, and Erda$_{GA}$ constitute a family of small programming languages and implementations for experimenting with error handling mechanisms. We use the unqualified name *Erda* to refer to the language family as a whole, or any one member of the family where the languages are all alike in relevant respects.

The "concrete" syntax of Erda resembles that of Racket (and Scheme).

# 1 Erda$_{RVM}$

`#lang erda/rvm` package: `erda`

The *Erda$_{RVM}$* language is a dynamically typed language that includes an *alerts* mechanism for declarative error reporting, and transparently propagates errors as data values.

This document describes the syntax and semantics of a selection of the Erda$_{RVM}$-specific constructs.

The `erda/rvm` language also inherits a number of constructs directly from Racket, including the `begin`, `begin0`, `let`, `let*`, `letrec`, `require`, and `provide` syntactic forms, and the `not` function. These forms should therefore behave as described in the Racket documentation. Note, however, that functions may seemingly behave differently due to Erda$_{RVM}$'s different function application semantics.

## 1.1 Modules and Macros

The Racket `require` and `provide` forms (and associated sub-forms) may be used in Erda as normal to import modules and to define the interfaces exposed by modules.

Macros are not included in the language by default, but there is nothing preventing from `require`'ing macro support from Racket.

## 1.2 Defining Forms

```
(define id expr)
(define (id arg ...) maybe-alerts expr ...+)
(define (id arg ...) #:handler maybe-alerts expr ...+)
(define (id arg ...) #:direct expr ...+)
```

Forms used to define variables and functions.

The semantics of the `(define id expr)` form is the same as in Racket. In Erda$_{RVM}$ functions are not first class, and the language does not include a `lambda` form, and thus this form is intended only for defining variables (that do not name functions).

The second form, which is for defining "regular" Erda$_{RVM}$ functions, binds `id` as a function that takes arguments `arg ...`. The language enforces that the arguments will all have to be good, wrapped values (i.e., values for which the predicate `good-result?` holds); there is an implicit alert guarding against bad values. Explicit alerts may be specified according to the `maybe-alerts` grammar. The result of the function should be a single wrapped value (i.e.,

values for which `result?` holds). Indeed, Erda$_{RVM}$ does not support multi-value returns, or more generally, multi-value expressions. The language enforces, on the single return value, the data invariant associated with its type; the function application will produce a bad value instead if the invariant does not hold. Explicit post-conditions are treated similarly. The invariants are not checked on bad results.

The `#:handler` variant of the `define` form is like the "regular" function definition form, but without the implicit alerts requiring good arguments, or the assumption that post-condition expressions require good free variables. That is, any pre-conditions get evaluated even if some of the arguments are bad, and if they hold, the function gets called. Similarly, any post-conditions (but not data invariant) are also checked on a bad result. The intention is for this kind of function definition to make it possible to implement "handler" functions able to process (and perhaps recover from) bad values.

The `#:direct` variant of the form defines a function that is called directly, without the language doing any pre- or post-processing on the incoming or outgoing values, which are still expected to be wrapped (no data invariants is checked either, so beware). This form of `define` is intended to allow for the implementation of Erda$_{RVM}$ functions (such that they are aware of wrapped values) as Racket-based primitives, probably using Racket's `#%plain-app` form as the "FFI" for implementing such primitives within the Erda$_{RVM}$ language. Perhaps more likely, you'll want to implement such functions in Racket, and instead merely `declare` them as `#:direct`.

```
(declare (id arg ...) maybe-alerts)
(declare (id arg ...) #:direct)
```

Forms used to specify information about functions, not to implement them, or to bind the identifier *id*. The binding must already exist.

The first `declare` form declares a Racket primitive that processes unwrapped values, and thus will get automatic unwrapping/wrapping at the application site. Alert clauses may be specified, with any `test-expr` evaluated with the *arg* (and `value`, as appropriate) identifiers bound to *wrapped* values; in other words, despite a primitive function being called, the conditional expressions are still written in Erda$_{RVM}$. As many existing Racket functions may throw exceptions, it is quite important to specify `on-throw` alert clauses as appropriate, as a way of converting from such a foreign error reporting mechanism.

The second `declare` form is like the `#:direct` `define` form, but without taking an implementation. An implementation must already be bound as the function *id*.

It is also possible to call undeclared Racket functions, as long as they are bound. Naturally, then, no explicit alerts have been specified, but goodness of arguments is nonetheless enforced, and arguments are unwrapped automatically; undeclared functions are expected to process unwrapped values. There is no catching of exceptions, but the data invariant of the result value is checked. A broken DI leads to the result being automatically wrapped as a bad value, whereas otherwise it is wrapped as a good value.

### 1.2.1 Alerts

The alert specification of a `defined` or `declared` function matches the following grammar.

```
maybe-alerts =
             | #:alert (alert-clause ...)

alert-clause = (alert-id pre-when test-expr)
             | (alert-id pre-unless test-expr)
             | (alert-id post-when test-expr)
             | (alert-id post-unless test-expr)
             | (alert-id on-throw pred-expr)
```

where:

*alert-id*

An alert name. The name of the alert to trigger if the corresponding `test-expr` holds, or if the corresponding `pred-expr` predicate holds for a thrown exception object.

*test-expr*

An Erda$_{RVM}$ expression, computing in wrapped values. The *test-expr*s of `#:handler` functions should probably be able to deal with bad values as well.

The expression is automatically negated for the `pre-unless` and `post-unless` cases.

For post-condition expressions, the result of the function application is bound as `value`.

*pred-expr*

A Racket predicate expression, computing in bare values. The predicate should accept any bare `exn` structure (or a subtype) as an argument, and yield a bare value indicating whether the predicate holds.

## 1.3 Expressions

`(#%datum . datum)`

A good literal value, as specified by *datum*.

For example:

```
> 0
(Good 0)
```

```
(quote id)
```

A good symbol *'id*. In Erda$_{RVM}$ symbols are primarily used to name alerts.

For example:

```
> 'not-found
(Good 'not-found)
```

```
value
```

A result value. Bound in the scope of a `define` or `declare` post-condition expression, for example, but also in some other syntactic contexts that have expressions for result processing (see `try`, for example).

```
(if test-expr then-expr else-expr)
```

Like Racket's `if`, but processes wrapped values. If *test-expr* yields a bad value, then the overall expression yields a bad value, and neither branch is evaluated. If *test-expr* yields a good `#f`, then the *else-expr* expression is evaluated. Otherwise *then-expr* is evaluated.

For example:

```
> (if (raise 'bad) 'yes 'no)
(Bad bad-arg monadic-if)←(Bad bad raise)
```

```
(if-not test-expr then-expr else-expr)
```

Equivalent to `(if` *test-expr else-expr then-expr*`)`.

```
(or expr ...)
```

Like Racket's `or`, but processes wrapped values. If any of the *expr*essions yields a bad value, none of the remaining expressions are evaluated, and the overall result will also be bad.

```
(and expr ...)
```

Like Racket's `and`, but processes wrapped values. If any of the *expr*essions yields a bad value, none of the remaining expressions are evaluated, and the overall result will also be bad.

```
(cond cond-clause ... else-clause)

cond-clause = (test-expr then-expr)

else-clause = (else then-expr)
```

A conditional expression that processes wrapped values. If any of the *test-expr*essions yields a bad value, none of the remaining expressions are evaluated, and the overall result will also be bad. Note the compulsory `else` clause, which is a significant difference compared to Racket's `cond`.

For example:

```
> (cond
    [#f 'false]
    [(raise 'bad) 'bad]
    [else 'otherwise])
(Bad bad-arg monadic-if)←(Bad bad raise)
```

```
(let-direct ([arg expr] ...) body ...+)
```

Locally switches to a "direct" computation mode so that the *body* expressions compute with bare values, without implicit error processing. Each argument value, given by *expr*, is unwrapped and bound to the corresponding *arg*, which must be an identifier. Said identifiers will be bound in the scope of the body. The result of the body expressions is then again wrapped, in either a good or bad wrapper, based on the DI. The overall expression fails if any *expr* yields a bad value, and in that case the body is left unevaluated.

```
(try body ...+ #:catch catch-clause maybe-catch-all)

   catch-clause = ((id ...) then-body ...+)

maybe-catch-all =
                | (_ then-body ...+)
```

Evaluates the *body* expressions, and if the last of them yields a bad result, then matches it against the *catch-clause*s based on the alert name of the bad value. The optional *maybe-catch-all* clause will match anything. If the body result is good, or if there is no matching clause, then that result remains the result of the overall expression. Otherwise the result is given by the last *then-body* expression of the first matching clause. Within a *catch-clause*, the bad value being handled is bound as `value`.

For example:

```
> (try (raise 'worst)
   #:catch [(bad worse still-worse) 1]
           [(worst) 2]
           [_ 3])
(Good 2)
> (try (raise 'bad) #:catch [(worse worst) 3])
(Bad bad raise)
> (try 1 #:catch [_ 3])
(Good 1)
> (try 1 #:catch [_ (define x 1) x])
(Good 1)
> (try (raise 'bad) #:catch [_ (bad-result? value)])
(Good #t)
> (try (raise 'bad) #:catch [_ 3])
(Good 3)
```

```
(::> try-expr ... fail-expr)
```

Evaluates the *try-expr* expressions in order, until one of them yields a good value, which then becomes the value of the overall expression. Where no *try-expr* evaluates to a good value, the result of the overall expression is that of *fail-expr*.

For example:

```
> (::> 'good 'alternative)
(Good 'good)
> (::> (raise 'bad) 'alternative)
(Good 'alternative)
> (::> (raise 'bad) (raise 'worse) 'alternative)
(Good 'alternative)
> (::> (raise 'bad) (::> (raise 'nested-bad)) (raise 'no-good))
(Bad no-good raise)
```

```
(on-alert (handler-clause ...) body ...+)

handler-clause = ((id ...) expr ...+)
```

Installs handlers for the scope of the *body* expressions, the last of which normally gives the result of the overall expression.

If any application of a function listed by *id* fails (with a bad result), then a matching clause's *expr*essions are evaluated, and the result of the last of them is substituted in place of the result of the failed function call.

This recovery mechanism does not apply to syntactic forms (even if named by *id*), nor will recovery happen within the body of an `let-direct` expression.

For example:

```
> (on-alert () 'nothing)
(Good 'nothing)
> (on-alert ([(not) 'good]) (raise 'bad))
(Bad bad raise)
> (on-alert ([(raise) 'good]) (raise 'bad))
(Good 'good)
```

```
(block stat ... result-expr)

stat = (#:let id expr)
     | (#:when test-expr #:let id expr)
     | expr
```

Evaluates a sequence of restricted "statements," in the order given. Each *stat* may be an assignment, a conditional assignment, or an expression. Conditional assignment only happens if the condition is a good true value. What appears to be assignment to a previously defined variable is actually a shadowing single static assignment. Any *id* that gets bound is in scope for the rest of the expression. A restriction of conditional assignment is that conditional assignment to an unbound *id* is not allowed, as then *id* might not be bound for the rest of the expression.

The evaluation of the overall expression immediately stops with a bad value if a *test-expr* produces a bad value. Where there were no failures in conditionals, the overall result of the expression will be that of *result-expr*.

For example:

```
> (block 1 2 3)
(Good 3)
> (block [#:let x 1] x)
(Good 1)
> (block [#:let x 1] [#:let x 2] x)
(Good 2)
> (block [#:let x 1] [#:when #t #:let x 2] x)
(Good 2)
> (block [#:let x 1] [#:when #f #:let x 2] x)
(Good 1)
> (block [#:let x (raise 'bad)] [#:when x #:let x 'good] x)
(Bad bad-arg monadic-if)←(Bad bad raise)
```

## 1.4 Standard Library

This section lists a small selection of the Erda$_{RVM}$ standard library.

The documented argument and result types (or predicates, rather) are only for informational purposes; they are not necessarily enforced using actual contracts (indeed Erda$_{RVM}$ does not have support for contracts built-in). Also, the contracts we use here are informal, in that we may mix and match Erda and Racket predicates.

Some functions do have pre- and post-conditions specified with alert clauses, but these are not indicated in the signatures shown here; the signatures here reflect the functions' own ability to handle inputs, as they have been implemented. The Erda$_{RVM}$ language itself does further bad-value extension, at call sites; this is different to Erda$_{GA}$, where it is the functions that are extended, and thus become more capable at dealing with bad values, and this difference is reflected in their documented contracts.

```
(result? x) → good-result?
  x : any/c
```

A predicate that holds if `x` is a wrapped value (whether good or bad). The result of the predicate is itself wrapped.

```
(good-result? x) → good-result?
  x : any/c
```

A predicate that holds if `x` is a good (wrapped) value.

```
(bad-result? x) → good-result?
  x : any/c
```

A predicate that holds if `x` is a bad (wrapped) value.

For example:

```
> (bad-result? (raise 'worst))
(Good #t)
```

```
(alert-name? x) → good-result?
  x : any/c
```

A predicate that holds if `x` is an alert name.

```
(raise alert-name) → bad-result?
  alert-name : alert-name?
```

Creates a new bad value with the specified `alert-name`, passed in as a wrapped symbol. The constructed bad value will have no history beyond the call to this function.

```
(raise-with-value alert-name v) → bad-result?
  alert-name : alert-name?
  v : result?
```

Creates a new bad value with the specified *alert-name*, and the specified value *v*, which was found to be unacceptable for some reason, giving raise to an alert. The value *v* may have good or bad wrapping.

```
(raise-with-cause alert-name cause) → bad-result?
  alert-name : alert-name?
  cause : result?
```

Creates a new bad value with the specified *alert-name* and the specified *cause*, where *cause* should be a badness that triggered the error being raised. The constructed bad value will have *cause* as a separate "branch" of history.

For example:

```
> (raise-with-cause 'follow-up (raise 'cause))
(Bad follow-up raise-with-cause)⇐(Bad cause raise)
```

# 2 Erda$_{RVM}{}^{\sigma}$

`#lang erda/sigma-rvm` package: `erda`

The *Erda$_{RVM}{}^{\sigma}$* language is a variant of Erda$_{RVM}$ such that it exports an assignment expression, and modified conditionals with optional cleanup actions. Only the additions are documented here.

Erda$_{RVM}{}^{\sigma}$'s `set!` form (for variable assignment) is the same as in Racket. Bad values also get assigned.

The only conditionals available in Erda$_{RVM}{}^{\sigma}$ are `if`, `when`, and `unless`. The other conditionals from Erda$_{RVM}$ are not available. More or less all other forms (e.g., `define`) and functions (e.g., `raise`) from Erda$_{RVM}$ *are* also included in Erda$_{RVM}{}^{\sigma}$.

```
(if test-expr then-expr else-expr maybe-cleanup)

maybe-cleanup =
              | #:cleanup cleanup-expr ...
```

Like Erda$_{RVM}$'s `if`, but may include cleanup actions. Said actions are given as a sequence of *`cleanup-expr`* expressions, which are evaluated for their side effects in the case that *`test-expr`* yields a bad value; this does not influence the result of the overall expression, which will still be as for Erda$_{RVM}$'s `if`.

For example:

```
> (define failed? #f)

> (if (raise 'worse) 1 2 #:cleanup (set! failed? #t))
(Bad bad-arg monadic-if/cleanup)←(Bad worse raise)
> failed?
(Good #t)
```

```
(when test-expr body ...+ maybe-cleanup)
```

Like Racket's `when`, but processes wrapped values, and may include cleanup actions. Where *`test-expr`* is a good `#f` value, the result of the overall expression will be `#<void>`, without any wrapper.

Note that there is no `when` or `unless` in Erda$_{RVM}$, as having them makes little sense without side effects (such as assignment).

For example:

```
> (when 1
    2 3)
(Good 3)
```

```
(unless test-expr body ...+ maybe-cleanup)
```

Like `when`, but evaluates *`body`* expressions in the case where *`test-expr`* is a good `#f` value.

For example:

```
> (unless 1
    2 3)

> (let ([x 1])
    (when (raise 'worst)
      (set! x 2)
      #:cleanup (set! x 3))
    x)
(Good 3)
```

# 3 Erda$_{C++}$

`#lang erda/cxx` package: `erda`

The *Erda*$_{C++}$ language is a statically typed language such that it includes an alerts mechanism for declarative error reporting, and transparently propagates errors as data values.

Erda$_{C++}$ is very similar to Erda$_{RVM}$, but with some notable differences:

- Provided that any referenced functions are implemented both for Racket and C++, the definitions appearing in a `erda/cxx` module (or collection thereof) may both be used directly from a Racket program, and translated into a C++ API and implementation usable from C++ programs. In contrast, the definitions appearing in a `erda/rvm` module are only intended for evaluation in the Racket VM.
- While the Erda$_{C++}$ and Erda$_{RVM}$ implementations are largely based on the same code, the former does somewhat more work at compile time. This is to keep the runtime requirements smaller, and thus facilitate translation into C++. The only C++ runtime requirements for the `erda/cxx` language itself are a small subset of the C and C++11 standard libraries, and the `"erda.hpp"` header file.
- Erda$_{C++}$'s functions and variables are typed, whereas in Erda$_{RVM}$ it is values that are typed. While the static types need not always be declared, the types of a program must be fully resolvable statically. For this purpose, the compiler features Hindley-Milner style type inference.
- For declaring types and other details relating to translating Erda$_{C++}$ into C++, the language features support for various annotations (e.g., `type`, `foreign`, etc.) that may be specified for declarations; there are no such annotations in Erda$_{RVM}$.
- Not everything from Erda$_{RVM}$ has been brought over to Erda$_{C++}$; notably, some of the error recovery supporting forms are missing, as is most of the runtime library. The focus in Erda$_{C++}$ has been to include only the essentials in the language, and exclude more experimental features (such as `try` and `on-alert`). The idea is to improve and validate the design of these features in Erda$_{RVM}$ first, before bringing them into other Erda variants.

This document describes the syntax and semantics a selection of those Erda$_{C++}$ constructs that have notable differences to Erda$_{RVM}$'s. Overall, Erda$_{C++}$'s syntactic constructs generally have the same semantics as in Erda$_{RVM}$, and we do not document them separately here.

```
(define #:type id maybe-annos)
(define id maybe-annos expr)
(define (id arg ...) maybe-annos maybe-alerts expr ...+)
(define (id arg ...) #:handler maybe-annos maybe-alerts expr ...+)
(define (id arg ...) #:direct maybe-annos expr ...+)
```

Forms used to define types, variables and functions.

These forms have the same semantics as for Erda$_{RVM}$'s `define`, with three notable exceptions. Firstly, there is a `define` `#:type` form, which is the same as for Magnolisp's `define`. Second, Erda$_{C++}$ does not support the `on-throw` alert clause; the *`maybe-alerts`* grammar is otherwise the same as given in the §1.2.1 "Alerts" section. Third, all the `define` variants accept optional annotations; the grammar for *`maybe-annos`* is as described in §3.2 "Annotations".

```
(declare #:type id maybe-annos)
(declare (id arg ...) maybe-annos maybe-alerts)
(declare (id arg ...) #:direct maybe-annos)
```

Forms used to specify information about types and functions, not to implement them, or to bind the identifier *`id`*. The binding must already exist.

See `define` for a description of the three notable differences between Erda$_{C++}$'s `declare` compared to Erda$_{RVM}$'s `declare` and Magnolisp's `declare`, as these differences are the same for both `define` and `declare`.

## 3.1 C++ Translation Advising Annotations

Some of the Erda$_{C++}$ defining forms support a subset of the annotations that appear in the Magnolisp language. The supported annotations are: `type`, `export`, `foreign`, and `literal`. The purpose of these annotations is to instruct Erda$_{C++}$-to-C++ translation. Refer to Magnolisp documentation for more details about them.

# 4 Erda$_{GA}$

`#lang erda/ga` package: `erda`

The *Erda*$_{GA}$ language is a dynamically typed functional language such that it includes an alerts mechanism for declarative error reporting, and transparently propagates errors as data values. A particular aim for Erda$_{GA}$ is to adhere to the semantics of guarded algebras.

Erda$_{GA}$ is superficially very similar to Erda$_{RVM}$, but it has some notable differences in both its design and implementation:

- In Erda$_{GA}$ functions are (wrapped) values, and hence may be passed around as such, enabling the definition of Erda$_{GA}$-native higher-order functions. An attempt to apply a bad function (or a non-function) will naturally fail, resulting in a bad value.
- Erda$_{GA}$ emits the necessary glue code for alert processing into function definition sites rather than call sites. Consequently, it is possible to support alerts also for anonymous functions.
- In Erda$_{GA}$, *all* function arguments are evaluated before determining whether they are good arguments for the function. Erda$_{RVM}$'s argument evaluation is less eager.
- Erda$_{GA}$ records failed expression history in a manner that allows for "replay" of failed operations, using `redo` and related functions.

This document describes the syntax and semantics a selection of those Erda$_{GA}$ constructs that have notable differences to Erda$_{RVM}$'s. Overall, Erda$_{GA}$'s constructs generally have the same syntax and semantics as those of Erda$_{RVM}$, and we do not document them separately here.

```
(declare (id arg ...) #:is tgt-id #:direct)
(declare (id arg ...) #:is tgt-id maybe-alerts)
```

Forms used to wrap a primitive target function *tgt-id* into a thunk that does error processing, and which is then bound as *id*.

The significant difference between Erda$_{GA}$ and Erda$_{RVM}$ is that since in Erda$_{RVM}$ error processing code is emitted into call sites, the same function can be applied in two modes, directly or with implicit error processing, with any alerts specified with Erda$_{RVM}$'s `declare`. In Erda$_{GA}$, on the other hand, "direct" and "Erda" functions have different bindings, and thus `let-direct` and related forms do not alter the error processing behavior of applications of `declare`d functions.

```
(lambda args #:direct body ...+)
(lambda args #:handler maybe-alerts body ...+)
(lambda (arg ...) #:primitive maybe-alerts body ...+)
```

```
(lambda args maybe-alerts body ...+)

args = (arg ...)
     | (arg ... . rest-id)
     | rest-id
```

An anonymous function expression.

A `#:direct` function must have a *body* that is prepared to handle wrapped (good or bad) argument values, and will produce a wrapped value.

A `#:handler` is not protected from receiving bad values as arguments, unless alerts are specified to guard against that.

The body of a `#:primitive lambda` is protected against bad arguments, and it is assumed that the body processes bare values, and produces a bare value. Any alerts that are given must be in terms of wrapped values, however.

The "regular" `lambda` form produces a function that processes wrapped values, but its *body* can expect to see no bad arguments.

For example:

```
> ((lambda (x) #:alert ([bad-arg pre-when #t]) x) 42)
(Bad bad-arg: <fun> 42)
> ((lambda (x) #:handler 42) (raise 'bad))
(Good 42)
```

```
(thunk rest ...+)
```

Like `lambda` without parameters.

```
(apply fun args) → Result?
  fun : Result?
  args : Result?
```

Applies the specified *fun*ction on the specified list of (wrapped) arguments *args*. (See the `args` functions for manipulating such lists.)

```
(function? x) → (Good/c rkt.boolean?)
  x : any/c
```

A predicate recognizing values that may be applied, e.g., with `apply`.

For example:

```
> (function? function?)
(Good #t)
```

```
(let-direct ([arg expr] ...) body ...+)
```

Behaves like Erda$_{RVM}$'s `let-direct`, except that the definition of function application is not altered as radically as for Erda$_{RVM}$.

```
(direct-lambda (arg ...) maybe-alerts body ...+)
```

Similar to `let-direct`, but wraps the direct code into an anonymous function, for which alerts may also be specified.

```
(define-direct (id arg ...) maybe-alerts body ...+)
```

A combination of `define` and `direct-lambda`.

```
(if-then cond-value then-thunk else-thunk) → Result?
  cond-value : Result?
  then-thunk : (Result/c rkt.procedure?)
  else-thunk : (Result/c rkt.procedure?)
```

A handler function used in translation of `if` forms. The *cond-value* determines which thunk gets applied, and it is acceptable for the unapplied one to be bad. This function is not meant to be used directly, but it may be useful to know its binding, in order to be able to compare function values when inspecting history.

For example:

```
> (if (raise 'bad) 1 2)
(Bad bad-arg: if-then (Bad bad: raise bad) <fun> <fun>)
```

```
(cond cond-clause ... else-clause)

cond-clause = (test-expr then-expr ...+)

else-clause = (#:else then-expr ...+)
```

A conditional expression that processes wrapped values. If any of the *test-expr*essions yields a bad value, none of the remaining expressions are evaluated, and the overall result will also be bad. Note the compulsory `#:else` clause, which is a significant difference compared to Racket's `cond`.

For example:

```
> (::> (cond
         [#f 'false]
         [(raise 'bad) 'bad]
         [#:else 'otherwise])
       'cond-was-bad)
(Good 'cond-was-bad)
```

```
(>>= v f) → Result?
  v : Result?
  f : Result?
```

An identity-monadic bind, such that *v* is passed directly onto the function *f* if both are good arguments for >>=. The argument *f* must be a function callable with one argument.

For example:

```
> (>>= #f not)
(Good #t)
> (>>= not #f)
(Bad bad-arg: >>= not #f)
```

```
(do sub-do ... expr)

sub-do = [x-id <- expr]
       | expr
```

A "monadic" expression that uses >>= as its monadic "bind" function, but gets its error-monadic semantics from the Erda$_{GA}$ language. Sequencing stops once an *expr*ession produces a bad value. Naturally, as Erdas process bad values throughout, any good first argument for >>= is not unwrapped, as a Haskeller might expect.

For example:

```
> (define bad (raise 'bad))

> (do 42)
(Good 42)
> (do bad 42)
(Bad bad-arg: >>= (Bad bad: raise bad) <fun>)
> (do [x <- bad] 42)
(Bad bad-arg: >>= (Bad bad: raise bad) <fun>)
```

```
(not v) → Result?
  v : Result?
```

Like Racket's `not`, but extended to process wrapped values.

There are still more commonalities in the two languages' syntax and standard library:

- Erda$_{GA}$'s `define` is like Erda$_{RVM}$'s `define`.
- Erda$_{GA}$'s `::>` is like Erda$_{RVM}$'s `::>`.
- Erda$_{GA}$'s `try` is like Erda$_{RVM}$'s `try`.
- Erda$_{GA}$'s `on-alert` is like Erda$_{RVM}$'s `on-alert`.
- Erda$_{GA}$'s `if` is like Erda$_{RVM}$'s `if`.
- Erda$_{GA}$'s `and` is like Erda$_{RVM}$'s `and`.
- Erda$_{GA}$'s `or` is like Erda$_{RVM}$'s `or`.
- Erda$_{GA}$'s `raise` is like Erda$_{RVM}$'s `raise`.
- Erda$_{GA}$'s `raise-with-cause` is like Erda$_{RVM}$'s `raise-with-cause`.

## 4.1 Inspecting and Using History

Erda$_{GA}$ has `result?`, `good-result?`, and `bad-result?` predicates, which are like Erda$_{RVM}$'s `result?`, `good-result?`, and `bad-result?`.

For inspecting the contents of bad values, Erda$_{GA}$ has functions such as `bad-result-alert-name` (which is also in Erda$_{RVM}$), `bad-result-fun`, and `bad-result-args`.

```
(bad-result-alert-name v) → (Result/c rkt.symbol?)
  v : Result?
```

Returns the alert name of `v` if it is a bad result. Otherwise returns a bad-argument error.

```
(bad-result-fun v) → Result?
  v : Result?
```

Returns the value whose application yielded the bad result `v`. (Such a value may not necessarily be a function, which would have caused a failure to apply it.) Returns a bad-argument error if `v` is not a bad result.

```
(bad-result-args v) → (Result/c rkt.list?)
  v : Result?
```

Returns the arguments of the failed operation that produced the bad result `v`, as an argument list containing wrapped values. (See the `args` functions for manipulating such lists.) Returns a bad-argument error if `v` is not a bad result.

```
(set-bad-result-args v args) → Result?
  v : Result?
  args : Result?
```

Functionally updates the bad result `v` to contain the argument list `args`. The length of the argument list should be compatible with any function recorded as `(bad-result-fun v)`.

```
(bad-result-args-map f v) → Result?
  f : Result?
  v : Result?
```

Applies function `f` to each argument in `(bad-result-args v)`, and then functionally updates the argument list of `v` to be the resulting list.

For example:

```
> (bad-result-args-map 1 2)
(Bad bad-arg: bad-result-args-map 1 2)
> (bad-result-args-map (lambda (x) #:handler (rkt.add1 x))
                       (bad-result-args-map 1 2))
(Bad bad-arg: bad-result-args-map 2 3)
> (bad-result-args-map (lambda (x) #:handler (rkt.add1 x))
                       (>>= 1 bad))
(Bad bad-arg: >>= 2 (Bad bad-arg: add1 (Bad bad: raise bad)))
```

For argument list manipulation Erda$_{GA}$'s standard library includes the functions `args-list?`, `args-list`, `args-cons`, `args-car`, `args-cdr`, and `args-list-set`, which are similar to the Racket equivalents, but defined as handlers as necessary to be able to deal with bad values contained in argument lists.

Erda$_{GA}$ is also able to "replay" failed operations with the `redo`, `redo-apply`, and `redo-app` functions, all of which are handlers so that they can accept a bad result as their first argument. The latter two `redo` functions make it possible to invoke the failed operation with different arguments.

```
(redo v) → Result?
  v : Result?
```

Replays the failed operation that produced the result `v`. It will naturally fail again with the same result unless it was dependent on state.

For example:

```
> (redo bad)
(Bad bad: raise bad)
```

```
(redo-apply v args) → Result?
  v : Result?
  args : Result?
```

Replays the failed operation that produced the result *v*, but using the specified argument list *args*.

For example:

```
> (redo-apply bad (args-list 'worse))
(Bad worse: raise worse)
```

```
(redo-app v arg ...) → Result?
  v : Result?
  arg : Result?
```

Replays the failed operation that produced the result *v*, but using the specified argument list *arg ...*.

For example:

```
> (redo-app bad 'still-worse)
(Bad still-worse: raise still-worse)
```

## 4.2 Contracts

```
(require erda/ga/contract)
```
package: erda

In documenting Erda$_{GA}$ functions, we state contracts for them as is usual for Racket documentation. These are stated so that they account for any bad-argument extension of the function, and are explicit about wrapped values. It is a *programming error* to call a function with arguments that do not adhere to the contract.

We specify the contracts for function arguments and results using predicates and predicate combinators, some of which come from Racket. Where we want to be explicit about our use of Racket primitives, we prefix them with `rkt.`.

Erda$_{GA}$ also has an internal API that includes some of its own predicates such as `Result?`, `Good?`, and `Bad?`, and some predicate combinators, which have a `/c` in their name, as is customary for contracts in Racket. These are only intended for internal use, documentation purposes, and perhaps for actually specifying enforced contracts.

We are presently not documenting alert declarations, which are different in their role. In particular, any bad conditions covered by declared alerts cannot be a programming error, since cases are handled implicitly.

```
(Result? x) → rkt.boolean?
  x : any/c
```

A predicate that recognizes Erda$_{GA}$'s wrapped values.

```
(Good? x) → rkt.boolean?
  x : any/c
```

A predicate that recognizes Erda$_{GA}$'s good wrapped values.

```
(Bad? x) → rkt.boolean?
  x : any/c
```

A predicate that recognizes Erda$_{GA}$'s bad wrapped values.

```
(Result/c p?) → (-> any/c rkt.boolean?)
  p? : rkt.procedure?
(Good/c p?) → (-> any/c rkt.boolean?)
  p? : rkt.procedure?
```

For building predicates on wrapped values, in terms of a *primitive* predicate that is passed in as an argument.

That is, calling `Result/c` or `Good/c` instantiates a Racket predicate which applies the specified Racket primitive predicate `p?` on any bare value inside a good value.

The difference between the two functions is that `Result/c` produces predicates that return `#t` for bad values, while `Good/c`'s predicates return `#f` for bad values.

# 5 Example Code

For sample Erda$_{RVM}$ code, see the `"i1-program.rkt"` file of the Erda implementation codebase. For Erda$_{GA}$ code, see `"i3-program.rkt"`, and the more extensive examples under `"tests/ga/test-*.rkt"`. Those programs should evaluate as is within the Racket VM; see the `racket` command of your Racket installation.

For sample Erda$_{C++}$ programs, see the `"test-*.rkt"` files and `"program-*"` projects in the `"tests"` directory of the codebase.

Most of the provided sample Erda$_{C++}$ programs will evaluate as is within the Racket VM. To instead translate said programs into C++, see the Magnolisp documentation, or look at the `"Makefile"`s in the `"program-*"` directories for example invocations of the `mglc` command-line tool.

To run basic tests to verify that the Magnolisp compiler is available and working, you may run:

```
make test
```

# 6 Source Code

The Erda source code repository is hosted at:

```
https://github.com/bldl/erda
```

# 7 Installation

The pre-requisites for installing the software are:

- **Racket.** The primary implementation language of Erda. Version 6.3 (or higher) of Racket is required; a known-compatible version is 6.5, but versions 6.3–6.6 are all expected to work.
- **Magnolisp.** A language and compiler serving as a basis for the implementation of Erda$_{C++}$. A known-compatible revision of Magnolisp is 191d529486e688e5dda2be677ad8fe3b654e0d4f.

The software and the documentation can be built from source, or installed using the Racket package manager. Racket is required for building and installation.

Once Racket and its tools have been installed, the Magnolisp and Erda packages can be installed with the `raco` commands:

```
raco pkg install git://github.com/bldl/magnolisp#191d529
raco pkg install git://github.com/bldl/erda
```

The above commands should install the library, the command-line tool(s), and a HTML version of the manual.

# 8 License

Except where otherwise noted, the following license applies:



Authors: Tero Hasu