Errors as Data Values as the Language Default [*]

Tero Hasu and Magne Haveraaen

Bergen Language Design Laboratory
Department of Informatics
University of Bergen, Norway
http://www.ii.uib.no/~{tero,magne}

Abstract

A "thrown" exception is a non-local side effect that complicates static reasoning about code. In some programs errors are instead propagated as ordinary values. Such propagation is sometimes done in monadic style, and some languages include syntactic conveniences for writing expressions in that style. We sketch a language-based failure management approach in which error-monad-resembling transparent error value propagation is made the language *default*. The approach accommodates language designs with all-referentially-transparent expressions, and syntactic conveniences resembling those of traditional exception mechanisms. Our proof-of-concept implementation of the approach is furthermore capable of automatically checking data invariants and function pre- and post-conditions, recording a trace of the failed or unevaluatable expressions caused by an error, and in some cases retaining "bad" values for potential use in recovering from an error.

1 Bad-Value-Extended Data Types as a Convention

A number of mainstream languages include a `try`/`catch`-style facility for intercepting non-local-returning, exceptional control transfers triggered by errors. Propagating error information using such mechanisms comes at the cost of making static reasoning about code harder. As exceptions are side effects, *referential transparency* (RT) of expressions that may throw is lost; this means that replacing an expression by its value may not preserve program semantics, as the value may not capture everything that the evaluation of an expression does.

A more traditional alternative is to report errors through return values, by having some values of the return type signify an error. While this approach preserves RT, it also tends to involve tedious, explicit checking and propagation of error return values. One may be able to encapsulate the tedious work within an abstraction, but the abstraction can only encompass operations for which there is a known way of determining which return values indicate an error. A simple way to enable such determination is to augment (in a general way) each return type to have a carrier set that is a disjoint union of values that are explicitly good or bad.

For example, for a potentially-failing Haskell function whose successfully computed values are of type `Integer`, we might specify the return type `Result Integer`, with the type constructor `Result` defined as shown below. In the definition, the parameter `t` is the type of the good "payload," and `Error` is the type of an error information object.

```
data Result t = Good t | Bad Error
```

`Result` can be made into an error monad by defining the operations of the `Monad` type class with the appropriate semantics. The monad then encapsulates implicit actions to check for `Good` function arguments, and to propagate any `Bad` arguments while skipping subsequent,

[*]This research has been supported by the Research Council of Norway through the project DMPL—Design of a Mouldable Programming Language.

unevaluatable function applications. It is not especially convenient to write monadic expressions such as `mv >>= (\x -> f x >>= (\y -> g y))`, but with Haskell's `do` notation as syntactic sugar one may instead write `do x <- mv; y <- f x; g y`. This is more convenient, but not uniform; the programmer has both monadic and "bare" values to deal with, and different syntaxes for monadic and non-monadic expressions (cf. the above expressions versus `g (f mv)`).

We note that the `Result` type constructor is universal: any type can be augmented with a set of error values by "wrapping" it in `Result`. What if we made use of this power for purposes of uniformity, and adopted a language-wide convention of computing with `Result`-wrapped values? We have been exploring this question, by devising and experimenting with various language-integrated mechanisms to support such a convention.

Two particularly useful, orthogonal kinds of such language-based support are transparent propagation of error values, and automated adaptation to other error reporting conventions, which we discuss in sections 1.1 and 1.2, respectively. Designing and implementing such support is made easier by consistency of convention and explicit badness of values: it is sufficient to only have "monadic" expression syntax, which can for the most part be made to look conventional, while still having it express `Result` processing; and generating code to check for a bad value requires no context, but merely the insertion of a single, known predicate check.

Our approach seems most promising for languages focused on supporting static reasoning, in that such languages benefit from RT preservation and simple, uniform rules for error management. Furthermore, static reasoning opportunities facilitate optimizations to reduce the readability-hampering code bloat that results from naive implementations of the approach.

We have implemented our language-integrated error management scheme in the form of Erda[1], which is a family of experimental programming languages that have a Racket-resembling syntax. Erda$_{RVM}$ is a dynamically typed language targeting the Racket virtual machine (VM), while Erda$_{C++}$ is a statically typed language that compiles to C++ source code. The implementation of Erda$_{RVM}$ is purely based on program transformations expressed in terms of the host language's macros, whereas Erda$_{C++}$ additionally relies on a compiler.

1.1 Uniform, Language-Wide Error Propagation

Assuming that every operation (whether user-defined or built-in) consistently reports any failures via distinctly bad return values, then what remains is to add language support for transparent processing of values that make such an explicit distinction. In our solution the processing includes checking for bad values, skipping operations that may not be performed, and recording information about failures and skipped operations inside the processed values, for the benefit of error reporting and recovery. The processing is performed by (highly portable) code that the language implementation emits between operations that appear in a program; that code is similar to the implicit actions encoded in terms of the primitives of an error monad.

In Erda$_{RVM}$, the literal syntax `0` expresses the constant value `(Good 0)`. The `factorial` function, as below, transparently processes `Result` values throughout; as shown here, it correctly obeys the convention of reporting errors (in this case its inability to compute $x!$ for any $x < 0$) by returning a `Bad` value, which can be instantiated with `raise`:

```
(define (factorial x)
  (cond
   [(< x 0) (raise 'bad-arg)]
   [(= x 0) 1]
   [else (* x (factorial (- x 1)))]))
```

[1]Code and documentation for Erda is available at `https://www.ii.uib.no/~tero/erda-2015/`

In addition to `raise`, Erda$_{RVM}$ includes a number of other constructs (e.g., a `try`/`catch`-inspired `try`, and a less familiar `on-alert` [1]) capable of dealing with `Bad` values. `Bad` values may be stored in variables as normal. By default, a `define`d function only receives `Good` arguments[2]; for example, the expression `(factorial (factorial -1))` only causes `factorial` to be called once, as Erda$_{RVM}$ skips the outer call due to its `Bad` argument.

In effect, Erda$_{RVM}$ has extended the implementation of `factorial` to allow the normal control flow of the program to proceed irrespective of whether the result is a good value or a bad value. If it is a good value, the computation will proceed as normal. If the result is a bad value, the computation can accumulate information about how the value should have been processed after its inception. In contrast, an error monad has no access to such contextual information; a language implementation can access even the uncomputable expression itself.[3]

1.2 Declarative Adaptation to Other Conventions

Our language-native error reporting convention concerns both natively defined functions and foreign primitives. Due to our wholesale adoption of the convention, we wanted to support a declarative way of statically generating adaptation code for interfacing with functions that follow other conventions. Our declaration-based abstraction over foreign conventions is modeled after *alerts*, as detailed by Bagge et al [1]. An alert may be triggered due to a broken pre- or post-condition or a thrown exception, as declared. Alerts are propagated as bad data values.

Racket functions can be called from Erda$_{RVM}$ directly, with automatic `Result` (un)wrapping of arguments and return values, but any associated alerts have to be `declare`d:

```
(declare (/ x y) #:alert ([div-by-0 on-throw exn:fail:contract:divide-by-zero?]))
```

The alert facility is also useful within Erda$_{RVM}$. For example, rather than invoking `raise` explicitly within `factorial`, we can make it conformant by declaring an alert for it:

```
(define (factorial x) #:alert ([bad-arg pre-when (< x 0)]) ———)
```

Guarded algebras [2] serve as the formal basis for our declarative checking of failure conditions. Formally, all the applicable data invariants (which may be defined for Erda$_{RVM}$ data types) and alert declarations together induce an idealized guard predicate for every expression. In practice we need not infer a complete pre-condition for an operation, but can rather have a code generator precede or surround or follow an operation invocation with all the appropriate individual guard expressions, in order to detect or catch errors. At the same time we can also insert code to ensure that the appropriate wrapper data type is used for the result, with available information about any error embedded into the wrapper.

References

[1] Anya Helene Bagge, Valentin David, Magne Haveraaen, and Karl Trygve Kalleberg. Stayin' alert: Moulding failure and exceptions to your needs. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE '06)*, pages 265–274, Portland, Oregon, October 2006. ACM Press.

[2] Magne Haveraaen and Eric G. Wagner. Guarded algebras: Disguising partiality so you won't know whether it's there. In *Recent Trends In Algebraic Development Techniques*, volume 1827 of *Lecture Notes in Computer Science*, pages 3–11. Springer-Verlag, 2000.

[2] Erda$_{RVM}$ also supports declaration of `#:handler` functions accepting `Bad` arguments.

[3] For a formal basis, one might consider `factorial` as defining a partial algebra for good arguments, with Erda$_{RVM}$ doing a free term completion of the algebra for the bad arguments.