# Errors as Data Values

Tero Hasu 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. Particularly in functional languages it is fairly common to instead propagate errors as ordinary values. The propagation is sometimes done in monadic style, and some languages include syntactic conveniences for writing expressions in that style. We discuss a guarded-algebra-inspired approach for integrating similar, implicit error propagation into a language with "normal" function application syntax. The presented failure management approach accommodates language designs with all-referentially-transparent expressions. It furthermore supports automatically checking data invariants and function pre- and post-conditions, recording a trace of any due-to-an-error unevaluateable or failed expressions, and in some cases retaining "bad" values for potential use in recovering from an error.

## 1 Introduction

Traditional error handling mechanisms include explicit checking and propagation of error return values, as well as `try`/`catch`-style language constructs for intercepting non-local-returning, exceptional control transfers triggered by errors. The return-value-based mechanism has the drawback of requiring extensive "manual code generation." The exception-throwing mechanism avoids that by transferring control over any code not capable of handling errors, but this comes at the cost of making static reasoning about code harder.

The difficulty of understanding errors and their propagation through code is the cause of much end user trouble. For example, an advanced editor for XML documents may provide an interactive scripting facility and a powerful substitution mechanism. A user may inadvertently provide a recursive substitution pattern causing the text buffer to overflow in the middle of an editing script. Let us consider three different possible outcomes (in increasing order of preference): ① the problem goes "undetected," causing inconsistencies in the editor's document representation, and resulting in a corrupted document when the panicked user hits the save button; ② the editor crashes, causing the user to lose the current edits; or ③ the editor detects the problem, undoes the effect of the script, but also presents the user with the text, the current edit script with all user

*This paper was presented at the NIK-2016 conference; see http://www.nik.no/.*

input, and an explanatory message of the problem. With current software practices the last outcome is utopia, and ② is the only outcome we could hope for if the programming language had a reasonable error mechanism. This paper presents an approach which can make ③ feasible.

One fundamental problem is that most error information is volatile, with errors having to be checked in their immediate reporting context (e.g., through error values or flags or preconditions) to avoid them going unnoticed. An unchecked error may also trigger a non-local transfer (or even termination) of control as a side effect. Observable side effects result in the loss of *referential transparency* [Str00], i.e., the property of expressions that in essence means that replacing a subexpression by its value preserves evaluation semantics. Losing this property hampers equational reasoning about programs, as equivalences between expressions become unclear. Furthermore, potentially throwing operations and `try`/`catch` harnesses can induce numerous control jump sources and targets, adding to static analysis complexity, with the interdependent nature of data and control flows as an aggravating factor [BS09, LMGH13].

Existing approaches from functional languages already show that referentially-transparent error-propagating sub-languages are possible to define. The `Either` type constructor in Haskell, for example, is sometimes used in defining types for function results which are *either* successfully computed, or contain information about a failure. As `Either` is also a `Monad` [Wad95], one can use Haskell's `do` notation to write an expression that implicitly detects and preserves any failure result of one of its sub-computations. The syntactic convenience of using monads is limited to `do` blocks, however, and there is no single error monad being consistently used for error reporting in Haskell (the language also features more traditional exceptions as a competing mechanism).

In this paper we explore the idea of making monadic-resembling error handling the default in a language, in a way that makes it possible to retain referential transparency of expressions. The handling is "monadic-resembling" not because the associated operations necessarily obey monad laws, but because wrapped values (with error information) and implicit actions between operations are involved, just as with error monads.

We furthermore explore a declarative language for adapting to different error reporting conventions. Operations need not be (re)programmed to report errors as wrapped values; rather, code for checking for errors and converting to "monadic" values can be inserted automatically. Our declarative language is based on the previously presented *alerts* [BDHK06, Bag12] error abstraction, designed to abstract over different concrete mechanisms; consequently, in our solution, pre- or post-conditions, exceptions, etc., can all trigger alerts, and be treated uniformly from an error handling point of view. Guarded algebras serve as the formal basis for our declarative checking of failure conditions.

For our error-information-enriched data types we have experimented with somewhat unorthodox information content, such that an error value may contain ① a history of expressions that failed to compute and ② any bad values produced by failed expressions. Both kinds of information may be useful in error recovery.

We have tried the presented error recording and propagation scheme with variants of a custom, small, purpose-built programming language named Erda.

## Contributions

The main contributions of this paper are:

- We show that abnormal control flows can be made implicit by realising them at the language level so that all the relevant language constructs check and process

additional ranges of data values. (We consider the idea more pervasively than in the case of `do` blocks and similar notations, limited to special constructs in a language.)

- We show that conversions between bare and wrapped values can be automated (where necessary, and without distinct syntax) based on declarative specifications, and explain how those specifications relate to guarded algebras.
- We suggest that the wrapped values might include a history of failed expressions (which failed to compute a "good" result), and show that such history can be useful for deciding on recovery actions. (It is common for a language to record a stack trace, or for a program to record a "cause" chain of errors, but here we consider recording information about the failed *expressions* and their arguments.)
- We suggest that wrapped values might also contain any "bad" values computed by unsuccessfully evaluated expressions (i.e., values that fail to fulfil a predefined invariant), and show that such values can also be useful for recovery.

## 2 Guarded Algebras

The theory of *guarded algebras* [HW00] is a systematic approach to handling partiality and error values in algebraic style specifications. The approach allows total models and partial models, e.g., non-termination, yet allows the simpler total algebra reasoning in both cases.

The basic idea of algebras is centred around signatures and models for the signature. A plain *signature* (interface declaration) is a set of atomic type declarations, including the type `predicate`, and a set of function and predicate declarations, where the arguments and results are from the set of types. A predicate is a function with the return type `predicate`. Every type has an equality predicate. A *model* or *algebra* for a signature provides:

- A set for each type, called the *carrier set*, where the type `predicate` has a carrier set containing at least {`TRUE`, `FALSE`}.
- A total set-theoretic function for each function declaration, where the argument and result sets are the corresponding carrier sets.

For every signature we get the notion of type correct expressions with typed formal variables.

A *guarded signature* is a signature such that it in addition provides:

- A predicate `good` for each type, separating the carrier's *good* (when the `good` predicate evaluates to `TRUE`) and *bad* values (otherwise). The carrier elements `TRUE` and `FALSE` for the `predicate` type are its good values.
- A function declaration may have an explicit guard, i.e., a predicate expression that identifies its good arguments (when it evaluates to `TRUE`), as a subset of the good values of its argument types.

There is a well-formedness constraint ensuring that a function cannot appear directly or indirectly in its own guard.

We get the plain version of a guarded signature by disregarding the guards for its function declarations. The expressions for a guarded signature are the same as those for its plain signature.

On a computer a type is defined by a data structure, and its carrier is the set of bit patterns admitted by the data structure. The good bit patterns are those we consider valid for a type, e.g., as given by a data invariant. For instance, for ISBN numbers or the Norwegian national identity number system (*fødselsnummer*) this means having valid checksums; for C and C++ unsigned integer types all bit combinations are valid; and for IEEE floating point numbers [IEE08] this means not being a `NaN` (*not a number*).[1]

The guard for a function can intuitively be thought of as its precondition. Normally we are only interested in how functions behave on the good values, and so every function gets an implicit guard ensuring that it only receives good values.

Some functions are *handler* functions, however, possessing the ability to recover from (some) errors: in addition to taking its good arguments to good values, a handler also takes some (or all) of the bad values of its argument data types to good values. For example, a handler might try to turn a bad value into a good one based on redundant information in the bad data, such as that of a fødselsnummer. The fødselsnummer is an 11 digit number designed to allow for error correction, i.e., it is possible to find the correct fødselsnummer from a number with a few errors of the kind humans often do when handling long digit sequences by hand (swapping digits, misrepresenting a digit, etc.).

In many cases the recovery action is dependent on the local context that causes the error. For instance, the two function definitions $r_1(x) = \sin(\pi x)/x$ and $r_2(x) = x/(\sin(\pi x) + x)$ both yield a `NaN` ($0/0$ error for $x = 0$), but in the former case we should recover to the value $\pi$, and in the latter to the value $1/(\pi + 1)$, as given by L'Hôpital's rule.

## Algebraic Specifications

An *algebraic specification* consists of a signature, giving the vocabulary, and a set of axioms. An *axiom* is a predicate expression, which *fails* for a model if it is `FALSE` for at least one combination of elements in the carriers of the model; otherwise the axiom *holds*. A model *satisfies* a specification if all axioms hold for the model.

A *guarded specification* is an algebraic specification with implicit *goodness* axioms. A goodness axiom ensures that the result of a function is a good value (by the type's `good` predicate) for its good arguments. Good arguments to a function are good values which also are accepted by its guard (and must be good values for every expression in the guarding predicate). Good values for an expression are values which maintain goodness through every subexpression of the expression.

A model *satisfies* a guarded specification when all axioms hold for all good arguments. Thus any bad argument is ignored by the specification. For instance, we can write a guarded axiom for division as `(a / b) * a = 1`, which automatically ignores the case when `b = 0` if the function `x/y` has `y` $\neq$ `0` as a guard. In the plain version we would need to explicitly exempt this case by writing `b` $\neq$ `0` $\Rightarrow$ `(a / b) * a = 1`.

The definition of satisfaction for guarded axioms ensures that they only restrict the behaviour of a model on the good arguments. The behaviour for bad arguments and bad values is not constrained by the axioms. The restriction to good arguments has a profound effect: axioms on guarded signatures can just ignore any problematic case, which might become overwhelming for plain specifications. Translating a guarded specification to a plain specification means ① removing the extra structure of the signature, ② making the implicit goodness axioms explicit (one per function, which must include checks for good

[1] For instance, a `NaN` is returned for the floating point operation $0/0$, but not for $0.1/0$, which instead yields a positive infinity value.

arguments), and ③ changing each axiom to include checks for good arguments for all its subexpressions.

Due to the nature of guarded algebras, writing out the implicit axioms does not add any information to the guarded specification. Actually, writing axioms with a goodness predicate does not add any constraint to a guarded algebra. Thus the goodness predicates can be kept hidden in a guarded signature, and only need to appear when we want to see the plain version of the signature and its specification.

If we use partial set-theoretic functions as models for function declarations, the above requirements on guarded algebras force the functions to be defined for all good arguments. A partial function may thus be undefined only for bad arguments.

## The Structure of the *Bad* Values

We define two guarded algebras to be *care-equivalent* [HW00] if they have the same sets of good carrier values and the same sets of good arguments for each function, and if the functions give the same results in both algebras for the good arguments. Thus care-equivalent algebras only differ in the selection of bad values, and the behaviour of the functions on bad arguments.

Studying the class of care-equivalent algebras, we get the following properties:

- This class has initial algebras. For the total case this is the free term extension. For the partial case this is the algebra where the functions are undefined for all bad arguments.
- The class has no final algebra. Specifically this means that there is no canonical way of dealing with a single error element as *the* bad value, nor is there a canonical way of avoiding bad values.

An initial algebra has a unique mapping to any algebra. Obviously the partial algebra that only defines good values and arguments has a unique inclusion to every algebra possibly defining bad values. The free term extension algebra informally represents the complete history of failed expressions. Every function call with a bad argument is an element of the bad values. Then of course every function call with such a bad value as argument is a bad value in itself, and so forth. This algebra is canonical in representing all information about what happened from the moment a bad argument was used, i.e., a script of what would have been done if the error had not occurred, while all good arguments will yield the expected computation.

A final algebra has a unique mapping from every algebra. In the class of care-equivalent algebras, an algebra may have functions that can recover from bad values. An algebra that recovers from a bad value does not have a mapping to an algebra that recovers from a different bad value, nor to an algebra that does not recover from a bad value—like the single error element algebra.

Handler functions computing good values from bad values have to obey the guarded specification on their good arguments, but are free to recover from bad arguments in any suitable way. In some cases they might replace the first bad arguments in the error history with good ones, and then run the script from the error history on the replacement values.

## Guarding Programs

An implementation for a plain signature defines an algebra for that signature, by means of the code defining the data structures (sets of bit patterns) for the types and the algorithms (computations on sets of bit patterns) for the function declarations.

When implementing a guarded signature, we need to provide defining code for the corresponding plain signature. Thus we need to provide a data structure and designate a `good` predicate for each type, and we need to provide code for each of the declared functions and predicates, including the `good` predicate for each type.

With this code in place, the implementation can be verified or tested for conformance with the guarded specification or its plain counterpart, both of which concern the implementation's behaviour on good values. Such validation includes the implicit axioms for the preservation of goodness (i.e., good arguments yield good results). This ensures *reliability*: the functions behave as specified on good arguments.

The guards for a function are more difficult to interpret when implementing the guarded signature. Technically, the plain version of a guard includes checks for good values for each argument, checks for good arguments for all subexpressions of the guards, and the actual guard predicates themselves.

In languages which support precondition declarations, such as Eiffel or Ada-SPARK, the plain version of the guard can be provided as a precondition. Otherwise, tool support for preconditions may be developed, e.g., JML for Java. In both cases, the tool can insert precondition checks wherever needed. If no tool support is available, precondition checks can be inserted manually at the start of every implemented function. Precondition checking can be avoided if the condition can be proven to hold. For instance, if the implementation satisfies the implicit goodness axioms, it is easy to prove that only external inputs need to be checked for goodness[2], as all internally produced values will be good values.

Traditionally, a precondition violation causes the program to terminate early, or otherwise disrupts the normal control flow. Alternatively, we can encode each such violation as a bad data value, and keep the normal control flow of the program, for referential transparency. The bad data values are automatically avoided by the guards, ensuring *robustness*: corrupt data does not induce uncontrolled behaviour, with possible security or safety issues for the application.

An implementer may decide to use a weaker precondition than the full plain condition corresponding to a guard. For instance, the binary search algorithm requires a sorted data array to function properly. Still, a program dealing with only mostly sorted data arrays might first attempt a binary search in the data, and on failing to find the key that way, revert to sequential search to make sure that the sought data has not been overlooked. In this scenario, the binary search function need not require a sorted array as a precondition; instead, its found item data type can admit an "item not found(key, data array)" bad value. Sequential search can then be done by a *handler* that passes a good value through, and recovers from an "item not found" by either finding the item and yielding the good item value, or by returning a good "no such item" value.

While we in some cases want to call a handler directly after a possibly failing function call, as we did for "item not found" recovery above, doing so later on in the program can also be useful. Section 1's XML editor, for instance, might behave as follows: An edit script execution yields a good buffer value if everything goes as expected, but can accumulate the offending edit script command and any subsequent edit commands (all with related arguments) into a bad value if something goes wrong. Whenever the editor goes back to a user input mode (in a good state), a function displays the buffer and awaits

[2] Checking of input data is very important, as corrupt data has been known to cause security problems; e.g., corrupt JPEG or TIFF images have caused arbitrary code execution in browsers and image processing software.

for user input. That function can be made into a handler that for a bad value (and the ensuing error history) displays the unaltered buffer and the failing command script with appropriate error messages, before awaiting user input: the user can then choose to modify the offending parts of the edit script before rerunning it, or issue some other command on the still consistent buffer. With this simple change in the code, we have realised the advanced recovery mechanism sought in case ③: The editor's error history is the free term extension—the initial bad values for the care-equivalent algebras. This demonstrates that the free term extension can be useful for generic error handling.

Many data types lack spare bit patterns to encode bad values; e.g., in C and C++ unsigned integers, all bit patterns correspond to valid numbers. A data type may also simply have too few spare patterns for encoding all the bad values we wish to differentiate; e.g., an extreme case of this is the free term extension of bad values. To cater for these situations we may need extensive changes to our data types, with corresponding upgrades for our functions.

# 3 Automatic Pervasive Error Handling

We have argued that we can achieve referentially transparent and pervasive error handling by injecting error conditions as bad values into a data type. In our approach, normal functions will ignore the bad data, and special handler functions can deal with it at appropriate places in the normal control flow.

## Automatic Bad Value Extensions

We define an *error extension* $\hat{T}_E$ of a normal type $T$ as a disjoint union data type[3] with *cases* `cgood`—containing only the good values of type $T$—and `cbad`—containing the error values $E$. The type $E$ is chosen according to what error information the programmer wants to record. Functions $f$ defined on $T$ will have to be extended to functions on $\hat{T}_E$, while obeying the same guarded axioms. With tool support, we will be able to almost transparently deal with error extensions from normal code on $T$.

**The Error History Extension** The base case of this extension is the free term extension, which defines the type $E$ to be all terms (abstract syntax trees) with bad arguments. Thus the carrier for $E$ is all terms representing expressions with argument values (encapsulated as terms) that cause evaluation errors, and all subsequent function calls having any such value as an argument.

Every function $f$ defined on the normal types $T$ gets extended to a function $\hat{f}$ on the encapsulated types $\hat{T}_E$.

$$\hat{f}(\hat{a}_1,\ldots,\hat{a}_k) \;=\; \begin{cases} \texttt{cgood}(f(a_1,\ldots,a_k)) & \text{if } \hat{a}_1 = \texttt{cgood}(a_1),\ldots,\hat{a}_k = \texttt{cgood}(a_k) \\ & \text{and } a_1,\ldots,a_k \text{ are good arguments for } f, \\ \texttt{cbad}(\text{`}f(\hat{a}_1,\ldots,\hat{a}_k)) & \text{otherwise.} \end{cases}$$

The notation ‘$e$ creates a term from the expression $e$. Note that $\hat{a}_i$ can be: ① a `cbad` value holding a part of the error history, which then gets extended by the $\hat{f}$ function call; ② an encapsulated value belonging to the `cgood` case yet forming a bad argument to the function $f$, in which case the entire expression becomes a `cbad` value containing the call and its argument values; or ③ an encapsulated value belonging to the `cgood` case and forming a good argument, in which case $f$ is evaluated and the result becomes an

[3] These are called algebraic data types in the context of functional languages.

encapsulated `cgood` value. The extended function $\hat{f}$ will obey the same guarded axioms as $f$.

An error history extension $E$ can also record additional information alongside the term, like the good or bad value from evaluating the guard, or a possible data invariant breaking value returned when calling $f$. This extra information is in principle reproducible from the error history, making the extended $E$ isomorphic to the free term extension. However, such extensions may simplify the implementation of handler functions (see below).

**A User-Defined Error Extension** A user-defined error extension is specified like the error history extension above, except that the user provides the type $E$, and defines a function `error_encoding` that will compute values in $E$ from the term $`f$ and an argument list $(\hat{a}_1, \ldots, \hat{a}_k)$. This can be useful if the errors for a type can be classified into a few cases. $E$ can then cover those cases, and nothing more, possibly saving significant resources over the full error history, yet providing some details about the error. In effect, the function `error_encoding` is a homomorphism from the error history extension to the user defined type $E$.

**A Singular Error Value Extension** This is the classical case where a type is extended with a single error value, the type $E$ becomes the singleton set, and the `cbad` case only indicates an error but no more detail about its origin.

**Handler Functions** A handler $\bar{f}$ is a function defined directly on $\hat{T}_E$, and as such it can recover from bad arguments while behaving as a normal function on the good cases. Handlers extend a function $f$ on $T$ with extra code for bad arguments, where $f$ is a normal function, or the identity function. Since the $\bar{f}$ extension handles the good arguments like the automatic extension $\hat{f}$, it will obey the guarded axioms for $f$. Handlers based on identity functions can be placed anywhere in a program's code without altering its reliability (as it will still obey the same guarded specification), but the handlers may radically improve the well-behavedness of the program on formerly bad input data.

How to best recover from errors may depend on the local context where the error appears; e.g., consider the functions $r_1$ and $r_2$ in the beginning of section 2. One way of recovering contextually is to place a specifically chosen handler function immediately after the expression that may cause an evaluation error. More generic handlers can be placed further away from the possible error sources, allowing a uniform handling of a larger set of errors; e.g., consider the XML editor discussed earlier.

*The Relationship between Bad Value Extensions and Monads*

A *monad* is an implementation of a certain kind of collection of operations [Wad95]. Monads serve as a popular way in functional programming to add sequentialisation, and to implement language with type-directed semantics. One popular application is to implement implicit checking for bad values.

An *error monad* is a data type that represents either a "good" value, or information about an error. Its monadic primitives apply monadic functions only on good values; once one of the functions returns a "bad" monadic value, the subsequent ones are skipped, and the bad value is the result of the combined computation. This has an effect similar to our Singular Error Value extension (although error type $E$ need not be a singleton set).

Monadic sequencing is not a natural fit for implementing error history accumulation in our scenario, however. In our case functions should appear total, enabling programming with "regular" syntax in "direct style" [Dan94], so that both good and bad results of subexpressions unconditionally share the same continuation. Thus, a natural strategy

for implementing history accumulation is by either modifying the semantics of function application, or by extending the applied functions with logic for bad value processing. Those are the approaches we use in our Erda language implementations.

## Bad Values from Run-Time Errors

In order to do an error extension $\hat{T}_E$ for functions (or predicates) $f$ on types $T$, we need to understand what error reporting mechanisms are used when calling $f$ on bad arguments from $T$. Hardware architectures and software engineering practices have many idiomatic ways of alerting software about such error information. We follow [BDHK06] in suggesting how we can translate the various alerting idioms systematically into our idiom of instantiating a `cbad` value in $E$.

**Preconditions** A precondition is a predicate defining the good arguments for a function. If the precondition predicate applied to the arguments does not yield `TRUE`, the arguments are bad and the term must be encapsulated as a `cbad` value.

**Postconditions** A postcondition is a predicate providing a specification of a function's result. A postcondition violation (the predicate does not yield `TRUE` on the result of $f$), can be interpreted as identifying bad arguments to $f$, hence the call should be encapsulated as a `cbad` value, possibly also including the computed violating result value. Such an approach is convenient if explicitly checking for good arguments is (almost) running the algorithm of $f$, while the postcondition predicate is a simple test on $f$'s result.

**Data invariants** A data invariant is a predicate provided with a data structure, where the predicate discerns between good values and inconsistent values. All values in `cgood` $T$ are assumed to yield `TRUE` when tested by $T$'s data invariant. The invariant is therefore assumed to be both an implicit precondition and an implicit postcondition for functions $f$, and can be treated as such for the purpose of creating $\hat{T}_E$ values. However, often the implementation of data invariant preserving functions may be split into helper functions. Helper functions may create an inconsistent value from a consistent or inconsistent value in $T$, or consume inconsistent $T$ values and yield a consistent value in $T$ for actual use. A helper function $g$ of the former kind will be encapsulated as a $\hat{g}$ function yielding (accumulated) $\hat{T}_E$ error values for postcondition violations. A helper function $h$ of the latter kind should be turned manually into a handler function $\bar{h}$ that takes $\hat{T}_E$ error values to $\hat{T}_E$ `cgood` values.

**Error codes** An error code is a good value outside of the normal return value range of a function, used by the function to signal an error. Error codes can be captured in the same way as postcondition violations.

**Error flags** Error flags are extra output arguments recording the state (OK, error, and possibly what kind of error) of the result of a function call. They behave like error codes, and need to be checked before the function's return value is used. Error flags can be captured as postcondition violations: the flag has to OK the returned data, otherwise the function received bad arguments.

**Status variables** Status variables are similar to error flags, except that they are global variables being set by the failing function. Normally a status variable is reset by the next function call, but sometimes the status variable is sticky and has to be explicitly reset. We can handle this by checking the status variable after each return, resetting it and treating its value similarly to the error flag. As an optimisation, sticky error flags allow the granularity of checking to be reduced.

**Exceptions** An exception mechanism causes a disruption of the normal control flow when an error occurs. In our setting, we need to catch any exception and translate it into

its corresponding `cbad` value. The computation can then follow the normal path for either case of $\hat{T}_E$, improving code reasoning, simplifying code transformation, etc.

# 4 Erda

The failure management solution described in this paper relies on specific support from the programming language used. We have implemented such language support for our proof-of-concept *Erda*[4] language family, whose most recent member is Erda$_{GA}$.

*Erda*$_{GA}$ (*Er*ror *da*ta *G*uarded *A*lgebras) is a custom, small, dynamically typed language implemented purely in terms of Racket's "normal" language implementation facilities, by macro expansion to Racket's core language, which is then executable directly in the Racket VM. In Racket, a language is normally implemented as a library whose exported macros define the surface syntax of the language [THSAC$^+$11].

Erda$_{GA}$ reuses Racket's module system directly, and bears a close syntactic resemblance to Racket. Despite the similar looking syntax, Erda$_{GA}$ is different in that most of its expressions take and yield wrapped values by default; all literals, for example, get an implicit `Good` wrapper. The wrapper data type is effectively an algebraic data type, with the `Good` and `Bad` variants defined as Racket structure types sharing a common, "abstract" `Result` supertype.

In Erda$_{GA}$, all uses of potentially-failing expression language get translated into function applications (e.g., `if` expressions translate to applications of an `if-then` function, with the "then" and "else" expressions as `lambda`-delayed arguments). This design choice allows the error history extension of section 3 (for functions) to be implemented throughout the language, essentially by just ensuring that function calls consistently obey the expected convention of processing good and bad values.

A function may be defined with Erda$_{GA}$'s `define` syntax, which accepts an optional `#:alert` clause for specifying guards for the function. The breaking of a function application's guard automatically results in a `Bad`-wrapped value, which makes it rarely necessary to explicitly construct bad values in code. The `#:alert` syntax closely resembles that of the the original alerts proposal [BDHK06]:

```
(define (div x y) #:alert ([div-by-0 pre-when (= y 0)])
  (rkt/ x y)) ;; call Racket's division operator `/`, assumed imported as `rkt/`
```

A `define`d function by default has an implicit guard enforcing all-good arguments. The optional `#:handler` modifier overrides this behaviour so that it becomes possible to define a function able to handle (some) bad arguments. For example, we can write a function that accepts *any* argument (even a `Bad` one), and returns `(Good 42)`:

```
(define (forty-two x) #:handler 42)
```

Erda$_{GA}$ features a transparent foreign function interface (FFI) mechanism, allowing Racket functions to be called directly, with automatic unwrapping of arguments and wrapping of results. It is not necessary to declare total Racket functions, but any partial ones should be `declare`d, using `#:alert` clauses to specify how errors are reported (so that, for example, an exception gets caught and converted to a `Bad` value):

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

In addition to the alert-supporting defining forms, Erda$_{GA}$ also includes language for constructing, inspecting, and modifying error values (and also for `redo`ing their history),

[4] `https://www.ii.uib.no/~tero/erda-nik16/`

allowing users to define their own handler functions for recovery or error reporting.

# 5 Related Work

IEEE floating point [IEE08] NaN (not-a-number) value handling is a prominent example of implicit bad value propagation: with "quiet" (as opposed to "signalling") NaN arguments, most operations produce a NaN result, which are encoded using bit patterns.

Rust[5] is a programming language featuring specific support for explicit error handling in terms of an `Either`-like wrapper data type. Rust has no support for implicit error processing based on declared information; on the contrary, it *encourages* programmers to check for errors by issuing warnings about unused `Result` values.

Swamy et al. [SGLH11] have presented a rewriting algorithm (for extending ML) to insert the necessary glue code to allow programming with monadic types $m\,\tau$ as if they were of the bare type $\tau$. The solution could be used to achieve implicit, pervasive error propagation for a program that is appropriately typed in terms of an error monad.

Kotelnikov [Kot14] has extended Scala with similar functionality for type-directed transformation, able to generate glue code not only for monads, but also other computation types. That makes it usable also with error applicative functors [MP08, section 5], for example. The implementation is as a macro, and only macro uses are transformed; the desired computation type must be explicitly specified for each use.

# 6 Conclusion

We have described a highly portable error reporting and propagation convention in which errors are represented as normal data values, and all operations are made to appear *total*; in effect, abnormal control is traded for abnormal data, achieving referential transparency also for code with run-time errors. The convention is made possible by uniformly extending all data types so that their values are distinctly either *good* or *bad*. Apart from less mechanical tasks such as implementing context-sensitive error recovery functions (or "handlers"), for example, code for adapting to the convention can be automatically generated from annotations (e.g., "alerts") on function declarations. Correct and pervasive adoption of the error handling convention ensures robustness of programs, which is an important prerequisite for security and safety.

By being free of side effects, the convention facilitates static program analysis. A formal basis for reasoning about the convention is provided by the theory of guarded algebras, which covers interface specifications, and reasoning about good values and arguments when types also admit bad data. A program can be validated against a guarded specification, to ensure the code's reliability.

An error recorded as a bad value can contain the entire failure history from the cause and up to a handler that inspects it, and takes appropriate recovery action. The recovery can be site specific, and applied immediately after a failing expression, as in the division-by-0 examples at the beginning of section 2; or, it might be performed by a generic handler that reports errors to the user, and allows him or her to intervene in further recovering and processing data, as demonstrated by our XML editor example towards the very end of section 2. Inserting error handlers becomes easier when ① they follow the normal control flow of data and not some exceptional control flow, and ② there is no need for explicit checking of error values at every call site. Pervasive and automatic treatment of errors as data values thus make recovering from failures significantly easier.

[5] `https://www.rust-lang.org/` (2016)

## Acknowledgments

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

# References

[Bag12] Anya Helene Bagge. Separating exceptional concerns. In *Proceedings of the 5th International Workshop on Exception Handling (WEH'12)*, 2012.

[BDHK06] 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)*, 2006.

[BS09] Martin Bravenboer and Yannis Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, pages 1–12, New York, NY, USA, 2009. ACM.

[Dan94] Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

[HW00] 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.

[IEE08] IEEE Computer Society. *IEEE Standard for Floating-Point Arithmetic (IEEE Std 754$^{TM}$-2008)*, August 2008.

[Kot14] Evgenii Kotelnikov. Type-directed language extension for effectful computations. In *Proceedings of the Fifth Annual Scala Workshop*, SCALA '14, pages 35–43, New York, NY, USA, 2014. ACM.

[LMGH13] Shuying Liang, Matthew Might, Thomas Gilray, and David Van Horn. Pushdown exception-flow analysis of object-oriented programs, 2013.

[MP08] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, January 2008.

[SGLH11] Nikhil Swamy, Nataliya Guts, Daan Leijen, and Michael Hicks. Lightweight monadic programming in ML. In *Proceedings of the ACM International Conference on Functional Programming (ICFP)*, 2011.

[Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1/2):11–49, 2000.

[THSAC$^{+}$11] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. *SIGPLAN Not.*, 47(6):132–141, June 2011.

[Wad95] Philip Wadler. Monads for functional programming. In *Advanced Functional Programming*, volume 925 of *LNCS*, pages 24–52, 1995.