# Functional Pearl: Do you see what I see?

???
???

## Abstract

A static type system is a compromise between rejecting all bad programs and approving all good programs, where *bad* and *good* are formalized in terms of a language's untyped operational semantics. Consequently, every useful type system rejects some well-behaved programs and approves other programs that go wrong at runtime. Improving the precision of a language's type system is difficult for everyone involved—designers, implementors, and users.

This pearl presents a simple, elaboration-based technique for refining the analysis of an existing type system; that is, we approve more good programs and reject more bad ones. The technique may be implemented as a library and requires no annotations from the programmer. A straightforward (yet immoral) extension of the technique is shown to improve the performance of numeric and vector operations.

## 1. The Spirit and Letter of the Law

Well-typed programs *do* go wrong. All the time, in fact:

```
Prelude> [0,1,2] !! 3
*** Exception: Prelude.!!: index too large
Prelude> 1 `div` 0
*** Exception: divide by zero
Prelude> import Text.Printf
Prelude Text.Printf> printf "%d"
*** Exception: printf: argument list ended prematurely
```

Of course, Milner's catchphrase was about preventing type errors. The above are all *value errors* that depend on properties not expressed by Haskell's standard list, integer, and string datatypes. Even so, it is obvious to the programmer that the expressions will go wrong and there have been many proposals for detecting these and other value errors [1, 3, 6]. What stands between these proposals and their adoption is the complexity or annotation burden they impose on language users.

Likewise, there are useful functions that many type systems cannot express. Simple examples include a `first` function for tuples of arbitrary size and a `curry` function for procedures that consume such tuples. The standard work-around [5] is to write size-indexed families of functions to handle the common cases, for instance:

```
Prelude> let curry_3 f = \ x y z -> f (x,y,z)
```

[Copyright notice will appear here once 'preprint' option is removed.]

This pearl describes a technique for statically detecting value errors and statically generalizing value-indexed functions. We catch all the above-mentioned wrong programs and offer a single implementation of `curry` that obviates the need to copy/paste and manage size-indexed versions. Furthermore, we demonstrate applications to regular expression matching, vectorized operations, and querying a database.

The key to our success–and also our weakness—is that we specialize procedure call sites based on compile-time constant values. Run-time input foils the technique, but nonetheless we have found the idea useful for many common programming tasks. Moreover, the approach may be implemented as a library and used as a drop-in fix for existing code. Simply importing the library overrides standard procedures with specialized ones. No further annotations are necessary; if specialization fails we default to the program's original behavior. Put another way, our technique interprets the *letter* of programs before the type system conducts its coarser, type-of-values analysis. Like Shakespeare's Portia, we understand that the phrase "pound of flesh" says nothing about drawing blood and specialize accordingly.

Our implementation happens to be for Typed Racket, but Typed Clojure, Haskell, OCaml, Rust, and Scala would have been equally suitable hosts. The main requirement is that the language provides a means of altering the syntax of a program before type checking. Such tools are more formally known as *macro* or *syntax extension* systems. At any rate, we sketch implementations for the five languages listed above in the conclusion.

Until that time when we must part, this pearl first describes our general approach in Section 2 and then illustrates the approach with specific examples in Section 3. We briefly report on practical experiences with our library in Section 4. Adventurous readers may enjoy learning about implementation details in Section ???, but everyone else is invited to skip to the end and try implementing a letter-of-values analysis in their language of choice.

***Lineage*** Herman and Meunier demonstrated how Racket macros can propagate information embedded in string values and syntax patterns to a static analyzer [7]. Their illustrative examples were format strings, regular expressions, and database queries. Relative to their pioneering work, our contribution is adapting Herman & Meunier's transformations to a typed programming language. By inserting type annotations and boolean guards, our transformations indirectly cooperate with the type checker without significantly changing the program's syntax. We also give a modern description of Racket's macro system and handle definitions as well as in-line constants.

***Eager Evaluation*** Our implementation is available as a Racket package. To install the library, download Racket and then run `raco pkg install ???`. The source code is on Github at: `https://github.com/???/???`. Suggestions for a new package name are welcome.

## 2. Interpretations and Translations

The out-of-bounds reference in `[0,1,2] !! 3` is evident from the definition of `!!` and the values passed to it. We also know that ``1 `div` 0`` will go wrong because division by zero is mathematically undefined. Similar reasoning about the meaning of `"%d"` and the variables bound in `\ x y z -> x` can determine the correctness of calls to `printf` and `curry`.

In general, our analysis begins with a class of predicates for extracting meta-information from expressions; for example the length of a list value or arity of a procedure.

$$[\![\mathit{interp}]\!] \; : \left\{\mathit{expr} \rightarrow \mathit{Maybe(value)}\right\}$$

Applying a function $\mathtt{f} \in [\![\mathit{interp}]\!]$ to a syntactically well-formed expression should either yield a value describing some aspect of the input expression or return a failure result.[1] Correct predicates `f` should recognize expressions with some common structure (not necessarily the expressions' type) and apply a uniform algorithm to computer their result. The reason for specifying $[\![\mathit{interp}]\!]$ over expressions rather than values will be made clear in Section 3.

Once we have predicates for extracting data from the syntax of expressions, we can use the data to guide program transformations. The main result of this pearl is defining a compelling set of such transformations.

$$[\![\mathit{transform}]\!] : \left\{\mathit{expr} \rightarrow \mathit{expr}\right\}$$

Each $\mathtt{g} \in [\![\mathit{transform}]\!]$ is a partial function such that $(\mathtt{g}\ \mathtt{e})$ returns either a specialized expression $\mathtt{e}'$ or fails due to a value error. These transformations should be applied to expressions $\mathtt{e}$ before type-checking; the critera for correct transformations can then be given in terms of the language's typing judgment $\vdash \mathtt{e} : \tau$ and untyped evaluation relation $\lfloor \mathtt{e} \rfloor \Downarrow \mathtt{v}$, where $\lfloor \mathtt{e} \rfloor$ is the untyped erasure of $\mathtt{e}$. We also assume a subtyping relation $<:$ on types.

- *If* $\vdash \mathtt{e} : \tau$ *and* $\vdash \mathtt{e}' : \tau'$ *then* $\tau' <: \tau$ *and both* $\lfloor \mathtt{e} \rfloor \Downarrow \mathtt{v}$ *and* $\lfloor \mathtt{e}' \rfloor \Downarrow \mathtt{v}$.
- *If* $\nvdash \mathtt{e} : \tau$ *but* $\vdash \mathtt{e}' : \tau'$ *then* $\lfloor \mathtt{e} \rfloor \Downarrow \mathtt{v}$ *and* $\lfloor \mathtt{e}' \rfloor \Downarrow \mathtt{v}$.
- *If* $\vdash \mathtt{e} : \tau$ *but* $\mathtt{e}' = \bot$ *or* $\nvdash \mathtt{e}' : \tau'$ *then* $\lfloor \mathtt{e} \rfloor \Downarrow \mathsf{wrong}$ *or diverges.*

If neither $\mathtt{e}$ nor $\mathtt{e}'$ type checks, then we have no guarantees about the run-time behavior of either term. The hope is that both diverge, but proving this fact in a realistic language is more trouble than it is worth.

Finally, we say that a translation $(\mathtt{g}\ \mathtt{e}) = \mathtt{e}'$ is *moral* if $\lfloor \mathtt{e} \rfloor$ is $\alpha$-equivalent to $\lfloor \mathtt{e}' \rfloor$. Otherwise, the tranlation has altered more than just type annotations and is *immoral*. All our examples in Section 1 can be implemented as moral translations. Immoral translations are harder to show correct, but also much more useful.

## 3. Real-World Metaprogramming

We have defined useful letter-of-values transformations for a variety of common programming tasks ranging from type-safe string formatting to constant-folding of arithmetic. These transformations are implemented in Typed Racket [8], which inherits Racket's powerful macro system [4].

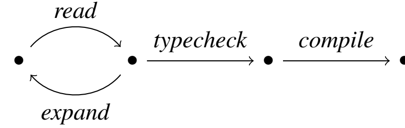


Figure 1: Typed Racket language model

Our exposition does not assume any knowledge of Racket or Typed Racket, but a key design choice of the Typed Racket language model bears mentioning: evaluation of a Typed Racket program happens across three distinct stages, shown in Figure 1. First the program is read and macro-expanded; as expanding a macro introduces new code, the result is recursively read and expanded until no macros remain. Next, the *fully-expanded* Typed Racket program is type checked. If checking succeeds, types are erased and the program is handed to the Racket compiler. For us, this means that we can implement $[\![\mathit{transform}]\!]$ functions as macros referencing $[\![\mathit{interp}]\!]$ functions and rely on the macro expander to invoke our transformations before the type checker runs.

Though we describe each of the following transformations using in-line constant values, our implementation applies $[\![\mathit{interp}]\!]$ functions to every definition and let-binding in the program and then associates compile-time data with the bound identifier. When a defined value flows into a function like `printf` without being mutated along the way, we retrieve this cached information. The macro system features used to implement this behavior are described in Section ???.

[1] The name *interp* is a mnemonic for *interpret* or *interpolant* [2].

## References


[1] Lennart Augustsson. Cayenne — a language with dependent types. In *Proc. ACM International Conference on Functional Programming*, pp. 239–250, 1998.

[2] William Craig. Three Uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. *Journal of Symbolic Logic* 22(3), pp. 269–285, 1957.

[3] W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. W. Schiller. Building and Using Pluggable Type Checkers. In *Proc. International Conference on Software Engineering*, 2011.

[4] Matthew Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. `http://racket-lang.org/tr1/`

[5] Daniel Friedlander and Mia Indrika. Do we need dependent types? *Journal Functional Programming* 10(4), pp. 409–415, 2000.

[6] Sam Lindley and Conor McBride. Hasochism: The Pleasure and Pain of Dependently Typed Programming. In *Proc. ACM SIGPLAN Notices*, pp. 81–92, 2014.

[7] David Herman and Philippe Meunier. Improving the Static Analysis of Embedded Languages via Partial Evaluation. In *Proc. ACM International Conference on Functional Programming*, 2004.

[8] Sam Tobin-Hochstadt and Matthias Felleisen. The Design and Implementation of Typed Scheme. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 395–406, 2008.