Functional Pearl: Do you see what I see?

???
???

Abstract

A static type system is a compromise between precision and usability. Improving the ability of a type system to distinguish correct and erroneous programs typically requires that programmers restructure their code or provide more type annotations, neither of which are desirable tasks.

This pearl presents an elaboration-based technique for refining the analysis of an existing type system on existing code without changing the type system. We have implemented the technique as a Typed Racket library. From the programmers' viewpoint, simply importing the library makes the type system more perceptive—no annotations or new syntax are required.

1. The Spirit and Letter of the Law

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

```
> (vector-ref (make-vector 2) 3)
==> vector-ref: index is out of range
> (/ 1 0)
==> /: division by zero
> (printf "~s")
==> printf: format string requires 1 argument
```

Of course, Milner's catchphrase was about preventing type errors. The above are all *value errors* that depend on properties not expressed by Typed Racket's standard vector, integer, and string datatypes. Even so, it is clear to the programmer that the expressions will go wrong.

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 [10] is to maintain size-indexed families of functions to handle the common cases, for instance:

```
> (define (curry_3 f)
    (λ (x) (λ (y) (λ (z) (f x y z)))))
```

These problems are well known, and are often used to motivate research on dependently typed programming languages [1]. Short of abandoning ship for a completely new type system, languages including Haskell, Java, OCaml, and Typed Racket have seen proposals for detecting certain values errors or expressing the polymorphism in functions such as `curry` with few (if any) changes to the existing type system [5, 11, 13, 19, 23]. What stands between these proposals and their adoption is the complexity or annotation burden they impose on language users.

This pearl describes a low-complexity, annotation-free (Section 4) technique for detecting value errors and expressing polymorphism over values. The key is to run a *textualist*[1] elaboration over programs before type-checking and propagate value information evident from the program syntax to the type checker. In terms of the first example in this section, our elaborator infers that the `vector-ref` at position `3` is out of bounds because it knows that `(make-vector n)` constructs a vector with `n` elements.

Our implementation is a Typed Racket library that shadows functions such as `make-vector` with textualist elaborators following the guidelines stated in Section 2. We make essential use of Racket's macro system [8] to reason locally, associate inferred data with bound identifiers, and cooperate with the rules of lexical scope. For the adventurous reader, Section 5 describes the main services provided by Racket's macros. Nevertheless, Typed Clojure [2], Rust [18], and Scala [17] could implement our approach just as well.

For a sense of the practical use-cases we envision, consider the function `regexp-match`, which matches a regular expression pattern against a string and returns either a list of matched substrings or `#false` if the match failed.

```
> (regexp-match #rx"(.*) v\\. (.*),"
    "Morrison v. Olson, 487 U.S. 654")
'("Morrison v. Olson" "Morrison" "Olson")
```

The parentheses in the regular expression delimit groups to match and return. In this example, there are two such groups. We have written an elaborator for `regexp-match` that will statically parse its first argument, count groups, and refine the result type of specific calls to `regexp-match`. The elaborator also handles the common case where the regular expression argument is a compile-time constant and respects α-equivalence.

Whereas Typed Racket will raise a type error on the following code because it cannot be sure `second` will produce a string, importing our library convinces Typed Racket that the code will succeed.

```
(define case-regexp #rx"(.*) v\\. (.*),")
(define rx-match regexp-match)

(define (get-plaintiff (s : String)) : String
  (cond
   [(rx-match case-regexp s)
    => second]
   [else "J. Doe"]))
```

Section 3 has more examples. Section 6 concludes.

[1] A textualist interprets laws by reading exactly the words on the page rather that by guessing the words' intended meaning.

Lineage Our technique builds on the approach of Herman and Meunier, who first demonstrated how Racket macros can propagate data embedded in string values and syntax patterns to a static analyzer [15]. Their illustrative examples were format strings, regular expressions, and database queries. Relative to their pioneering work, we adapt Herman and Meunier's transformations to a typed language, suggest new applications, and describe how to compose the results of analyses.

2. Interpretations, Elaborations

A textualist elaborator (henceforth, *elaborator*) is a specific kind of macro, meant to be run on the syntax of a program before the program is type-checked. The behavior of an elaborator is split between two functions: interpretation and elaboration.

An *interpretation* function attempts to parse data from an expression; for example parsing the number of groups from a regular expression string. In the Lisp tradition, we will use the value `#false` to indicate failure and refer to interpretation functions as *predicates*. Using `expr` to denote the set of syntactically valid, symbolic program expressions and `val` to denote the set of symbolic values, we define the set $\mathcal{I}$ of interpretation functions.

$$\mathcal{I} : \left\{ \texttt{expr} \to (\texttt{val} \cup \texttt{\#false}) \right\}$$

If `f` $\in \mathcal{I}$ and `e` $\in$ `expr`, it may be useful to think of `(f e)` as *evidence* that the expression `e` is recognized by `f`. Alternatively, `(f e)` is a kind of interpolant [3], representing data embedded in `e` that justifies a certain program transformation. Correct interpretation functions `f` obey two guidelines:

- The expressions for which `f` returns a non-`#false` value must have some common structure.
- Non-`#false` results `(f e)` are computed by a uniform algorithm and must have some common structure.

This vague notion of common structure may be expressible as a type in an appropriate type system. It is definitely not a type in the target language's type system.

Functions in the set $\mathcal{E}$ of *elaborations* map expressions to expressions, for instance replacing a call to `curry` with a call to `curry_3`. We write elaboration functions as $[\![\cdot]\!]$ and their application to an expression `e` as $[\![\texttt{e}]\!]$. Elaborations are allowed to fail raising syntax errors, which we notate as $\bot$.

$$\mathcal{E} : \left\{ \texttt{expr} \to (\texttt{expr} \cup \bot) \right\}$$

The correctness specification for an elaborator $[\![\cdot]\!] \in \mathcal{E}$ is defined in terms of the language's typing judgment $\vdash \texttt{e} : \tau$ and evaluation relation $\texttt{e}_\flat \Downarrow \texttt{v}$. The notation $\texttt{e}_\flat$ is the untyped erasure of `e`. We also assume a subtyping relation $<:$ on types. Let $[\![\texttt{e}]\!] = \texttt{e'}$:

- *If* $\vdash \texttt{e} : \tau$ *and* $\vdash \texttt{e'} : \tau'$
  then $\tau' <: \tau$
  and both $\texttt{e}_\flat \Downarrow \texttt{v}$ *and* $\texttt{e'}_\flat \Downarrow \texttt{v}$.
- *If* $\nvdash \texttt{e} : \tau$ *but* $\vdash \texttt{e'} : \tau'$
  then $\texttt{e}_\flat \Downarrow \texttt{v}$ *and* $\texttt{e'}_\flat \Downarrow \texttt{v}$.
- *If* $\vdash \texttt{e} : \tau$ *but* $\texttt{e'} = \bot$ *or* $\nvdash \texttt{e'} : \tau'$
  then $\texttt{e}_\flat \Downarrow \texttt{wrong}$ *or* $\texttt{e}_\flat$ *diverges.*

If neither `e` nor `e'` type checks, then we have no guarantees about the run-time behavior of either term. In a perfect world both would diverge, but the fundamental limitations of static typing [6] and computability keep us imperfect.

At present, these correctness requirements must be checked manually by the author of a function in $\mathcal{I}$ or $\mathcal{E}$.

2.1 Cooperative Elaboration

Suppose we implement a currying operation $[\![\cdot]\!]$ such that e.g. $[\![$`(curry (λ (x y) x))`$]\!]$ = `(curry_2 (λ (x y) x))`. The arity of `(λ (x y) x)` is clear from its representation. The arity of the result could also be derived from its textual representation, but it is simpler to add a *tag* such that future elaborations can retrieve the arity of `(curry_2 (λ (x y) x))`.

Our implementation uses a tagging protocol, and this lets us share information between unrelated elaboration function in a bottom-up recursive style. Formally speaking, this changes either the codomain of functions in $\mathcal{E}$ or introduces an elaboration environment mapping expressions to values.

3. What we can learn from Values

We have defined useful elaborators for a variety of common programming tasks ranging from type-safe string formatting to constant-folding of arithmetic. These elaborators are implemented in Typed Racket [22], a macro-extensible typed language that compiles into Racket [9]. An important component of Typed Racket's design is that all macros in a program are fully expanded before type-checking begins. This protocol lets us implement our elaborators as macros that expand into typed code.

Each elaborator is defined as a *local* transformation on syntax. Code produced by an elaboration may be associated with compile-time values for other elaborators to access. Using these values, we support variable bindings and propagate information upward through nested elaborations.

Conventions

- Interpretation and elaboration functions are defined over symbolic expressions and values; specifically, over *syntax objects*. To distinguish terms and syntax objects, we quote the latter and typeset it in green. Hence `(λ (x) x)` is the identity function and `'(λ (x) x)` is a syntax object.
- Values are typeset in `green` because their syntax and term representations are identical.
- Syntax objects carry lexical information, but our examples treat them as flat symbols.
- We use an infix `::` to write explicit type annotations and casts, for instance `(x :: Integer)`. Annotations and casts normally have two different syntaxes, respectively `(ann x Integer)` and `(cast x Integer)`.

3.1 Format Strings

Format strings are the world's second most-loved domain-specific language (DSL). All strings are valid format strings; additionally, a format string may contain *format directives* describing *where* and *how* values can be spliced into the format string. Racket follows the Lisp tradition of using a tilde character (`~`) to prefix format directives. For example, `~s` converts any value to a string and `~b` converts a number to binary form.

```
> (printf "binary(~s) = ~b" 7 7)
==> binary(7) = 111
```

If the format directives do not match the arguments to `printf`, most languages fail at run-time. This is a simple kind of value error that could be caught statically.

```
> (printf "binary(~s) = ~b" "7" "7")
==> printf: format string requires an exact-number
```

Detecting inconsistencies between a format string and its arguments is straightforward if we define an interpretation `fmt->types` for reading types from a format string value. In Typed Racket this function is rather simple because the most common directives accept `Any` type of value.

`fmt->types` $\in \mathcal{I}$

```
> (fmt->types "binary(~s) = ~b")
==> '[Any Integer]
> (fmt->types '(λ (x) x))
==> #false
```

Now to use `fmt->types` in an elaboration. Given a call to `printf`, we check the number of arguments and add type annotations using the inferred types. For all other syntax patterns, we perform the identity elaboration.

$[\![\cdot]\!] \in \mathcal{E}$

```
> [['(printf "~a")]]
==> ⊥
> [['(printf "~b" 2)]]
==> '(printf "~b" (2 :: Integer))
> [['printf ]]
==> 'printf
```

The first example is rejected immediately as a syntax error. The second is a valid elaboration, but will lead to a static type error. Put another way, the format string `"~b"` specializes the type of `printf` from `(String Any * -> Void)` to `(String Integer -> Void)`. The third example demonstrates that higher-order uses of `printf` default to the standard, unspecialized behavior.

3.2 Regular Expressions

Regular expressions are often used to capture sub-patterns within a string.

```
> (regexp-match #rx"-(2*)-" "111-222-3333")
==> '("-222-" "222")
> (regexp-match #rx"¥(.*)" "$2,000")
==> #false
```

The first argument to `regexp-match` is a regular expression pattern. Inside the pattern, the parentheses delimit sub-pattern *groups*, the dots match any single character, and the Kleene star matches zero-or-more repetitions of the pattern-or-character preceding it. The second argument is a string to match against the pattern. If the match succeeds, the result is a list containing the entire matched string and substrings corresponding to each group captured by a sub-pattern. If the match fails, `regexp-match` returns `#false`.

Groups may fail to capture even when the overall match succeeds. This can happen when a group is followed by a Kleene star.

```
> (regexp-match #rx"(a)*(b)" "b")
==> '("b" #f "b")
```

Therefore, a catch-all type for `regexp-match` is fairly large: `(Regexp String -> (U #f (Listof (U #f String))))`. Using this general type is cumbersome for simple patterns where a match implies that all groups will successfully capture.

```
> (define (get-domain (email : String)) : String
    (cond
     [(regexp-match #rx"(.*)@(.*)" email)
      => third]
     [else "Match Failed"]))
==> Type Error: expected String, got (U #false String)
```

We alleviate the need for casts and guards in simple patterns with a parentheses-counting interpretation that parses regular expressions and returns the number of groups. If there is any doubt whether a group will capture, we default to the general `regexp-match` type.

`rx->groups` $\in \mathcal{I}$

```
> (rx->groups #rx"(a)(b)(c)")
==> 3
> (rx->groups #rx"((a)b)")
==> 2
> (rx->groups #rx"(a)*(b)")
==> #false
```

The corresponding elaboration inserts casts to subtype the result of calls to `regexp-match`. It also raises syntax errors when an uncompiled regular expression contains unmatched parentheses.

$[\![\cdot]\!] \in \mathcal{E}$

```
> [['(regexp-match #rx"(a)b" str)]]
==> '((regexp-match #rx"(a)b" str)
      :: (U #f (List String String)))
> [['(regexp-match "(" str)]]
==> ⊥
```

3.3 Anonymous Functions

By tokenizing symbolic λ-expressions, we can statically infer their domain.

`fun->domain` $\in \mathcal{I}$

```
> (fun->domain '(λ (x y z)
                  (x (z y) y)))
==> '[Any Any Any]
> (fun->domain '(λ ([x : Real] [y : Real])
                  x))
==> '[Real Real]
```

When domain information is available at calls to a `curry` function, we elaborate to a type-correct, indexed version of `curry`. Conceptually, we give `curry` the unusable type `(⊥ -> ⊤)` and elaboration produces a subtype `curry_i`.

$[\![\cdot]\!] \in \mathcal{E}$

```
> [['(curry (λ (x y) x))]]
==> '(curry_2 (λ (x y) x))
```

This same technique can be used to implement generalized `map` in languages without variable-arity polymorphism [21].

```
> (define (cite (p : String) (d : String))
    (printf "~a v. ~a, U.S.\n" p d))
> (define plaintiff*
    '("Rasul" "Chisholm"))
> (define defendant*
    '("Bush" "Georgia"))
> (map cite plaintiff* defendant*)
==> Rasul v. Bush, U.S.
    Chisholm v. Georgia, U.S.
```

Leaving out an argument to `printf` or passing an extra list when calling `map` will raise an arity error during elaboration. On the other hand, if we modified `cite` to take a third argument then the above call to `map` would raise a type error.

3.4 Numeric Constants

The identity interpretation `id` $\in \mathcal{I}$ lifts values to the elaboration environment. When composed with a filter, we can recognize types of compile-time constants.

`int? ∘ id = int?` $\in \mathcal{I}$

```
> (int? 2)
==> 2
> (int? "~s")
==> #false
```

Constant-folding versions of arithmetic operators are now easy to define in terms of the built-in operations. Our implementation reuses a single fold/elaborate loop to make textualist wrappers over `+`, `expt` and others.

$[\![\cdot]\!] \in \mathcal{E}$

```
> (define a ⟦3⟧)
> (define b ⟦(/ 1 (- a a))⟧)
==> Error: division by zero
```

Partial folds also work as expected.

```
> (* 2 3 7 z)
==> '(* 42 z)
```

Taken alone, this re-implementation of constant folding in an earlier compiler stage is not very exciting. But since folded expressions propagate their result upwards to arbitrary analyses, we can combine these elaborations with a size-aware vector library to guard against index errors at computed locations.

3.5 Fixed-Length Structures

Fixed-length data structures are often initialized with a constant or computed-constant length. Racket's vectors are one such structure. For each built-in vector constructor, we thus define an interpretation:

$\texttt{vector->size} \in \mathcal{I}$

```
> (vector->size '#(0 1 2))
==> 3
> (vector->size '(make-vector 100))
==> 100
> (vector->size '(make-vector (/ 12 3)))
==> 4
```

After interpreting, we associate the size with the new vector at compile-time. Other elaborators can use and propagate these sizes; for instance, we have implemented elaborating layers for 13 standard vector operations. Together, they constitute a length-aware vector library that serves as a drop-in replacement for existing code. If size information is ever missing, the operators silently default to Typed Racket's behavior.

$[\![\cdot]\!] \in \mathcal{E}$

```
> ⟦(vector-ref (make-vector 3) (+ 2 2))⟧
==> ⊥
> ⟦(vector-length (vector-append '#(A B) '#(C D)))⟧
==> 4
> ⟦(vector-ref (vector-map add1 '#(3 3 3)) 0)⟧
==> (unsafe-ref (vector-map add1 '#(3 3 3)) 0)
```

For the most part, these elaborations simply manage sizes and delegate the main work to Typed Racket vector operations. We do, however, optimize vector references to unsafe primitives and specialize operations like `vector-length`, as shown above.

3.6 Database Schema

Racket's `db` library provides a string-based API for executing `sql` queries.[2] After connecting to a database, `sql` queries embedded in strings can be run to retrieve row values, represented as Racket vectors. Queries may optionally contain *query parameters*—natural numbers prefixed by a dollar sign (`$`). Arguments substituted for query parameters are guarded against `sql` injection.

```
> (define C (sql-connect #:user "admin"
                         #:database "SCOTUS"))
> (query-row C
    "SELECT plaintiff FROM rulings
     WHERE name = '$1' LIMIT 1"
    2001)
==> #("Kyllo")
```

This is a far cry from language-integrated query [14] or Scala's LMS [16], but the interface is relatively safe and very simple to use.

[2] In this section, we use `sql` as an abbreviation for `postgresql`.

Typed Racket programmers may use the `db` library by assigning type signatures to functions like `query-row`. This is somewhat tedious, as the distinct operations for querying one value, one row, or a lazy sequence of rows need similar types. A proper type signature might express the database library as a functor over the database schema, but Typed Racket does not have functors or even existential types. Even if it did, the queries-as-strings interface makes it impossible for a standard type checker to infer type constraints on query parameters.

The situation worsens if the programmer uses multiple database connections. One can either alias one query function to multiple identifiers (each with a specific type) or weaken type signatures and manually type-cast query results.

3.6.1 Phantom Types To the Rescue

By associating a database schema with each connection, our elaboration technique can provide a uniform solution to these issues. We specialize both the input and output of calls to `query-row`, helping catch bugs as early as possible.

Our solution, however, is not entirely annotation-free. We need a schema representing the target database; for this, we ask the programmer to supply an S-expression of symbols and types that passes a `schema?` predicate. This approach is similar to phantom types [12].

$\texttt{schema?} \in \mathcal{I}$

```
> (define scotus-schema
    '([decisions [(id . Natural)
                  (plaintiff . String)
                  (defendant . String)
                  (year . Natural)]]))
> (schema? scotus-schema)
==> '([decisions ....])
```

The above schema represents a database with at least one table, called `decisions`, which contains at least 4 rows with the specified names and types. In general a schema may specify any number of tables and rows. We statically disallow access to unspecified ones in our elaborated query operations.

In addition to the `schema?` predicate, we define one more interpretation and two elaborations. The first elaboration is for connecting to a database. We require a statically-known schema object and elaborate to a normal connection.

$[\![\cdot]\!] \in \mathcal{E}$

```
> ⟦'(sql-connect #:user "admin"
                 #:database "SCOTUS")⟧
==> ⊥
> ⟦'(sql-connect scotus-schema
                 #:user "admin"
                 #:database "SCOTUS")⟧
==> '(sql-connect #:user "admin"
                  #:database "SCOTUS")
```

The next interpretation and elaboration are for reading constraints from query strings. We parse `SELECT` statements using `sql->constr` and extract

- the names of selected columns,
- the table name, and
- an association from query parameters to column names.

$\texttt{sql->constr} \in \mathcal{I}$

```
> (sql->constr "SELECT * FROM loans")
==> '[* decisions ()]

 > (sql->constr "SELECT defendant FROM decisions
                 WHERE plaintiff = '$1'")
 ==> '[(defendant) decisions ($1 . plaintiff)]
```

The schema, connection, and query constraints now come together in elaborations such as a wrapper $[\![\cdot]\!]$ over `query-row`. There is still a non-trivial amount of work to be done resolving wildcards and validating row names before the type-annotated result is produced, but all the necessary information is available, statically.

$[\![\cdot]\!] \in \mathcal{E}$

```
> [[(query-row C
      "SELECT plaintiff FROM decisions
       WHERE year = '$1' LIMIT 1"
     2006)]]
==> ((query-row C
        "SELECT ...."
        (2006 : Natural))
      ::   (Vector String))

> [[(query-row C
      "SELECT * FROM decisions
       WHERE plaintiff = '$1' LIMIT 1"
      "United States")]]
==> ((query-row C
        "SELECT ..."
        ("United States" : String))
      ::   (Vector Natural String String Natural))

> [[(query-row C "SELECT foo FROM decisions")]]
==> ⊥
```

Results produced by `query-row` are vectors with a known length; as such they cooperate with our library of vector operations described in Section 3.5. Accessing a constant index into a row vector is statically guaranteed to be in bounds.

4. Backwards Compatibility

Our initial experience programming with the library has been positive. By far the most useful application is to `regexp-match`, as Typed Racket's default requires either a type cast or guards on each matched group. The bugs reported by `printf` and others have also proven useful in development.

To gauge the applicability of our library to existing code, we have applied it to 10,000 lines of Typed Racket code taken from 7 small projects. In total, we had to modify 6 lines to replace unrestricted mutation–which could violate our library's tagging assumptions—with reference cells.[3] This gives us confidence that retroactively opting-in to our library truly is a 1-line effort. Using the library also enables the removal of casts and type annotations made redundant by our elaborations.

Compiling with our library adds no statistically significant overhead, but tends to produce slightly larger bytecode files due to the inserted annotations (at most 2% larger). Running times we observed were on largely unaffected, but one project exhibited a 2-second slowdown due to added type casts. This could be improved by a closer integration with the type checker to remove casts guaranteed to succeed. Overall though, we find these performance characteristics encouraging.

5. More than a Pretty Face

Figure 1 gives a few statistics regarding our implementation. The purpose of this section is to explain why the line counts are low.

In total, the code for our six applications described in Section 3 comprise 901 lines of code (LOC). Another 145 lines implement common functionality, putting the grand total just over 1000 LOC.

[3] None of the surveyed code used the database library; we have only tested that interface in scripts.

Module	LOC	$\mathcal{I}$ (LOC)	$\mathcal{E}$ (LOC)
db	263	2 (78)	2 (101)
format	66	1 (33)	1 (21)
function	56	1 (5)	2 (27)
math	90	1 (3)	5 (46)
regexp	137	6 (60)	5 (33)
vector	228	1 (19)	13 (163)
Total	901	12 (204)	27 (415)

Figure 1: Quantifying the implementation

Except for db and regexp, each of the core modules defines a single interpretation function (in $\mathcal{I}$). In db, the two functions are the schema predicate and sql query parser. In regexp, we have six group-parsing functions to match the six string-like input types accepted by Racket's `regexp-match`. These group parsers, however, share a 33-line kernel. Incidentally, the average size of all value-interpreting functions is 33 LOC. The smallest interpreter is the composition of Racket's `number?` predicate with the identity interpretation in math (3 LOC). The largest is the query parser (35 LOC), though the analyses for format strings and regular expressions are approximately the same size.

The elaboration functions are aliases for standard library procedures. Typically, these functions match a syntactic pattern, check for value errors, and elaborate to a specialized Typed Racket procedure call. All these tasks can be expressed concisely; the average size of a function in $\mathcal{E}$ is 10 lines and the median is 7 lines. Much of the brevity is due to amortizing helper functions, so we include helpers' line counts in the figure.

5.1 Elegant Elaborations

Our elaboration for `vector-length` is straightforward. If called with a size-annotated vector `v`, `(vector-length v)` elaborates to the size. Otherwise, it defaults to Typed Racket's `vector-length`. The implementation is equally concise, modulo some notation.

```
(make-alias #'vector-length
  (syntax-parser
   [(_ v:vector/length)
    #''v.evidence]
   [_ #false]))
```

- `(make-alias id f)` creates a elaboration from an identifier `id` and a partial function `f`.
- The symbol `#'` creates a syntax object from a value or template.
- A `syntax-parser` is a match statement over syntactic patterns. This parser recognizes two cases: application to a single argument via the pattern `(_ v:vector/length)` and anything else with the wildcard `_`.
- The colon character (`:`) used in `v:vector/length` binds the variable `v` to the *syntax class* `vector/length`.
- The dot character (`.`) accesses an *attribute* of the value bound to `v`. In this case, the attribute `evidence` is set when the class `vector/length` successfully matches the value of `v`.

The pattern `v:vector/length` unfolds all elaborations to `v` recursively. So, as hinted in Section 3.5, we handle each of the following cases as well as any other combination of length-preserving vector operations.

```
> (vector-length #(0 1 2))
2
> (vector-length (vector-append #(A B)
                                #(C D)))
```

```
4
```

The structure of `vector-length` is common to many of our elaborations: we define a rule to handle an interesting syntactic pattern and then generate an alias from the rule using the helper function `make-alias`.

```
(define ((make-alias orig-id elaborate) stx)
  (or (elaborate stx)
      (syntax-parse stx
       [_:id
        orig-id]
       [(_ e* ...)
        #`(#,orig-id e* ...)])))
```

The elaboration defined by `(make-alias id elaborate)` is a function on syntax objects. This function first applies `elaborate` to the syntax object `stx`. If the result is not `#false` we return. Otherwise the function matches its argument against two possible patterns:

- `_:id` recognizes identifiers with the built-in syntax class `id`. When this pattern succeeds, we return the aliased `orig-id`.
- `(_ e* ...)` matches function application. In the result of this branch, we declare a syntax template with `#‘` and splice the identifier `orig-id` into the template with `,#`. These operators are formally known as `quasisyntax` and `unsyntax`; you may know their cousins `quasiquote` and `unquote`.

The identifier `...` is not pseudocode. In a pattern, it captures zero-or-more repetitions of the preceding pattern—in this case, the variable `e*` binds anything so `(_ e* ...)` matches lists with at least one element.[4] All but the first element of such a list is then bound to the identifier `e*` in the result. We use `...` in the result to flatten the contents of `e*` into the final expression.

A second example using `make-alias` is `vector-ref` $\in \mathcal{E}$, shown below. When given a sized vector `v` and an expression `e` that expands to a number `i`, the function asserts that `i` is in bounds. If either `vector/length` or `expr->num` fail to coerce numeric values, the function defaults to Typed Racket's `vector-ref`.

```
(make-alias #'vector-ref
  (syntax-parser
   [(_ v:vector/length e)
    (let ([i (expr->num #'e)])
      (if i
        (if (< i (syntax->datum #'v.evidence))
          #`(unsafe-vector-ref v.expanded '#,i)
          (raise-vector-bounds-error #'v i))
        #false))]
   [_ #false]))
```

This elaboration does more than just matching a pattern and returning a new syntax object. Crucially, it compares the *value* used to index its argument vector with that vector's length before choosing how to expand. To access these integer values outside of a template, we lift the pattern variables `v` and `e` to syntax objects with a `#'` prefix. A helper function `expr->num` then fully expands the syntax object `#'e` and the built-in `syntax->datum` gets the integer value stored at the attribute `#'v.evidence`.

Programming in this style is similar to the example-driven explanations we gave in Section 3. The interesting design challenge is making one pattern that covers all relevant cases and one algorithm to uniformly derive the correct result.

[4] The variable name `e*` is our own convention.

Syntax Class	Purpose
fun/domain	Infer function domain types
num/value	Evaluate a numeric expression
pattern/groups	Count regexp groups
query/constr	Parse `sql` queries
schema/spec	Lift a schema value from syntax
string/format	Parse format directives
vector/length	Infer length from a vector spec.

Figure 2: Registry of syntax classes

5.2 Illustrative Interpretations

Both `id` and `vector/length` are useful syntax classes.[5] They recognize syntax objects with certain well-defined properties. In fact, we use syntax classes as the front-end for each function in $\mathcal{I}$. Figure 2 lists all of our syntax classes and ties each to a purpose motivated in Section 3.

The definitions of these classes are generated from predicates on syntax objects. One such predicate is `vector?`, shown below, which counts the length of vector values and returns `#false` for all other inputs. Notice that the pattern for `make-vector` recursively expands its first argument using the `num/value` syntax class.

```
(define vector?
  (syntax-parser #:literals (make-vector)
   [#(e* ...)
    (length (syntax->datum #'(e* ...)))]
   [(make-vector n:num/value)
    (syntax->datum #'n.evidence)]
   [_ #f]))
```

From `vector?`, we define the syntax class `vector/length` that handles the mechanical work of macro-expanding its input, applying the `vector?` predicate, and caching results in the `evidence` and `expanded` attributes.

```
(define-syntax-class vector/length
 #:attributes (evidence expanded)
 (pattern e
  #:with e+  (expand-expr #'e)
  #:with len (vector? #'e+)
  #:when (syntax->datum #'len)
  #:attr evidence #'len
  #:attr expanded #'e+))
```

The `#:attributes` declaration is key. This is where the earlier-mentioned `v.expanded` and `v.evidence` properties are defined, and indeed these two attributes form the backbone of our protocol for cooperative elaborations. In terms of a pattern `v:vector/length`, their meaning is:

- `v.expanded` is the result of fully expanding all macros and elaborations contained in the syntax object bound to `v`. The helper function `expand-expr` triggers this depth-first expansion.
- `v.evidence` is the result of applying the `vector?` predicate to the expanded version of `v`. In general, `v.evidence` is the reason why we should be able to perform elaborations using the value bound to `v`.

If the predicate `vector?` returns `#false` then the boolean `#:when` guard fails because the value contained in the syntax object `len` will be `#false`. When this happens, neither attribute is bound and the pattern `v.vector/length` will fail.

[5] The name `vector/length` should be read as "vector *with* length information".

5.3 Automatically Handling Variables

When the results of an elaboration are bound to a variable `v`, we frequently need to associate a compile-time value to `v` for later elaborations to use. This is often the case for calls to `sql-connect`:

```
(define C (sql-connect ....))
(query-row C ....)
```

Reading the literal variable `C` gives no useful information when elaborating the call to `query-row`. Instead, we need to retrieve the database schema for the connection bound to `C`.

The solution starts with our implementation of `sql-connect`, which uses the built-in function `syntax-property` to associate a key/value pair with a syntax object. Future elaborations on the syntax object `#'(sql-connect e* ...)` can retrieve the database schema `#'s.evidence` by using the key `connection-key`.

```
(syntax-parser
 [(_ s:schema/spec e* ...)
  (syntax-property
    #'(sql-connect e* ...)
    connection-key
    #'s.evidence)]
 [_ (raise-syntax-error 'sql-connect
      "Missing schema")])
```

Storing this syntax property is the job of the programmer, but we automate the task of bubbling the property up through variable definitions by overriding Typed Racket's `define` and `let` forms. New definitions search for specially-keyed properties like `connection-key`; when found, they associate their variable with the property in a local hashtable whose keys are α-equivalence classes of identifiers. New `let` bindings work similarly, but redirect variable references within their scope. The technical tools for implementing these associations are `free-id-table`s and `rename-transformer`s (Section 5.4.6).

5.4 Ode to Macros: Greatest Hits

This section is a checklist of important meta-programming tools provided by the Racket macro system. Each sub-section title is a technical term; for ease of reference, our discussion proceeds from the most useful tool to the least.

5.4.1 Syntax Parse

The `syntax/parse` library [4] provides tools for writing macros. It provides the `syntax-parse` and `syntax-parser` forms that we have used extensively above. From our perspective, the key features are:

- A rich pattern-matching language; including, for example, repetition via `...`, `#:when` guards, and matching for identifiers like `make-vector` (top of Section 5.2) that respects α-equivalence.
- Freedom to mix arbitrary code between the pattern spec and result, as shown in the definition of `vector-ref` (bottom of Section 5.1).

5.4.2 Depth-First Expansion

Racket's macro expander normally proceeds in a breadth-first manner, traversing an AST top-down until it finds a macro in head position. After expansion, sub-trees are traversed and expanded. This "lazy" sort of evaluation is normally useful because it lets macro writers specify source code patterns instead of forcing them to reason about the syntax trees of expanded code.

Our elaborations, however, are most effective when value information is propogated bottom up from macro-free syntactic values through other combinators. This requires depth-first macro expansion; for instance, in the first argument of the `vector-ref` elaboration defined in Section 5.1. Fortunately, we always know which positions to expand depth-first and Racket provides a function `local-expand` that will fully expand a target syntax object. In particular, all our syntax classes listed in Figure 2 locally expand their target.

5.4.3 Syntax Classes

A syntax class encapsulates common parts of a syntax pattern. With the syntax class shown at the end of Section 5.2 we save 2-6 lines of code in each of our elaboration functions. More importantly, syntax classes provide a clean implementation of the ideas in Section 2. Given a function in $\mathcal{I}$ that extracts data from core value/expression forms, we generate a syntax class that applies the function and handles intermediate binding forms. Functions in $\mathcal{E}$ can branch on whether the syntax class matched instead of parsing data from program syntax.

In other words, syntax classes provide an interface that lets us reason locally when writing elaborators. The only question we ask during elaboration is whether a syntax object is associated with an interpreted value—not how the object looks or what sequence of renamings it filtered through.

5.4.4 Identifier Macros

Traditional macros may appear only in the head position of an expression. For example, the following are illegal uses of the built-in `or` macro:

```
> or
==> or: bad syntax
> (map or '((#t #f #f) (#f #f)))
==> or: bad syntax
```

Identifier macros are allowed in both higher-order and top-level positions, just like first-class functions. This lets us transparently alias built-in functions like `regexp-match` and `vector-length` (see Section 5.1). The higher-order uses cannot be checked for bugs, but they execute as normal without raising new syntax errors.

5.4.5 Syntax Properties

Syntax properties are the glue that let us compose elaborations. For instance, `vector-map` preserves the length of its argument vector. By tagging calls to `vector-map` with a syntax property, our system becomes aware of identities like:

```
(vector-length (vector-map f v))
  ==
(vector-length v)
```

Furthermore, syntax properties place few constraints on the type of data used as keys or values. This proved useful in our implementation of `query-row`, where we stored an S-expression representing a database schema in a syntax property.

5.4.6 Rename Transformers, Free Id Tables

Cooperating with `let` and `define` bindings is an important usability concern. To deal with `let` bindings, we use a `rename-transformer`. Within the binding's scope, the transformer redirects references from a variable to an arbitrary syntax object. For our purposes, we redirect to an annotated version of the same variable:

```
> ⟦'(let ([x 4])
      (+ x 1))⟧
==> '(let ([x 4])
      (let-syntax ([x (make-rename-transformer #'x
                        secret-key 4)])
        (+ x 1)))
```

For definitions, we use a *free-identifier table*. This is less fancy–it is just a hashtable whose keys respect α-equivalence–but still useful in practice.

5.4.7 Phasing

Any code between a `syntax-parse` pattern and the output syntax object is run at compile-time to generate the code that is ultimately run. In general terms, code used to directly generate run-time code executes at *phase level* 1 relative to the enclosing module. Code used to generate a `syntax-parse` pattern may be run at phase level 2, and so on up to as many phases as needed [7].

Phases are explicitly separated. By design, it is difficult to share state between two phases. Also by design, it is very easy to import bindings from any module at a specific phase. The upshot of this is that one can write and test ordinary, phase-0 Racket code but then use it at a higher phase level. We also have functions like `+` available at whatever stage of macro expansion we should need them—no need to copy and paste the implementation at a different phase level [23].

5.4.8 Lexical Scope, Source Locations

Perhaps it goes without saying, but having macros that respect lexical scope is important for a good user and developer experience. Along the same lines, the ability to propogate source code locations in elaborations lets us report syntax errors in terms of the programmer's code rather than locations inside our library. Even though we may implement complex transformations, errors can always be traced to a line number in the source.

6. Closing the Books

This pearl described a class of macros called *textualist elaborators* designed to enhance the analysis of an existing type system without any input from programmers. The main idea is to interpret values from the source code of a program and elaborate the same code into a form that helps the type system see what is obvious to a human reader. From an engineering perspective, we used tools provided by Racket's macro system to enable local, compositional, and lexically-scoped reasoning when designing new elaborators.

A key hypothesis of this pearl is that our framework could be implemented in a variety of languages using their existing type and syntax extension systems. Typed Clojure's macros [2], Rust's compiler plugins [18], and Scala's LMS framework [17] seem especially well-suited to the task. Template Haskell [20] and OCaml ppx[6] could also reproduce the core ideas, albeit after the programmer modifies function call-sites.

Whereas types support reasoning independent of data representations, we have argued that at least the source code details of values are worth taking into account during type checking. Doing so will catch more errors and enable unconventional forms of polymorphism, all without changes to the language's type system or the programmer's code.

Acknowledgments

To appear

References

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

[2] Clojure 1.8.0. Macros. 2016. `http://clojure.org/reference/macros`

[3] 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.

[4] Ryan Culpepper and Matthias Felleisen. Fortifying Macros. In *Proc. ACM International Conference on Functional Programming*, pp. 235–246, 2010.

[5] 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.

[6] Mike Fagan. Soft Typing: An Approach to Type Checking for Dynamically Typed Languages. PhD dissertation, Rice University, 1992.

[7] Matthew Flatt. Composable and Compilable Macros: You Want it When? In *Proc. ACM International Conference on Functional Programming*, pp. 72–83, 2002.

[8] Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that Work Together: Compile-Time Bindings, Partial Expansion, and Definition Contexts. In *Proc. Journal Functional Programming*, pp. 181–216, 2012.

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

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

[11] Andrew Kent, David Kempe, and Sam Tobin-Hochstadt. Occurrence Typing Modulo Theories. In *Proc. ACM Conference on Programming Language Design and Implementation*, 2016.

[12] Daan Leijen and Erik Meijer. Domain Specific Embedded Compilers. In *Proc. ACM Conference on Domain-specific languages*, pp. 109–122, 1999.

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

[14] Erik Meijer, Brain Beckman, and Gavin Bierman. LINQ: Reconciling Object, Relations and XML in the .NET Framework. In *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 706–706, 2006.

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

[16] Tiark Rompf and Nada Amin. Functional Pearl: A SQL to C compiler in 500 Lines of Code. In *Proc. ACM International Conference on Functional Programming*, pp. 2–9, 2015.

[17] Tiark Rompf, Nada Amin, Adriaan Moors, Philipp Haller, and Martin Odersky. Scala-Virtualized: Linguistic Reuse for Deep Embeddings. *Higher-Order and Symbolic Programming* 25(1), pp. 165–207, 2012.

[18] Rust 1.7. Compiler Plugins. 2016. `https://doc.rust-lang.org/book/compiler-plugins.html`

[19] Oleg Kiselyov and Chung-chieh Shan. Lightweight static capabilities. In *Proc. ACM Workshop Programming Languages meets Program Verification*, 2006.

[20] Tim Sheard and Simon Peyton Jones. Template Meta-programming for Haskell. In *Proc. Haskell Workshop*, 2002.

[21] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical Variable-Arity Polymorphism. In *Proc. European Symposium on Programming*, pp. 32–46, 2009.

[22] 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.

[23] Richard A. Eisenberg and Stephanie Weirich. Dependently Typed Programming with Singletons. In *Proc. Haskell Workshop*, pp. 117–130, 2012.

[6] `http://whitequark.org/blog/2014/04/16/a-guide-to-extension-points-in-ocaml/`