Type Tailoring

Ashton Wiersdorf
University of Utah, USA

Stephen Chang
UMass Boston, USA

Matthias Felleisen
Northeastern University, USA

Ben Greenman
University of Utah, USA

Abstract

Type systems evolve too slowly to keep up with the quick evolution of libraries—especially libraries that introduce abstractions. Type tailoring offers a lightweight solution by equipping the core language with an API for modifying the elaboration of surface code into the internal language of the typechecker. Through user-programmable elaboration, tailoring rules appear to improve the precision and expressiveness of the underlying type system. Furthermore, type tailoring cooperates with the host type system by expanding to code that the host then typechecks. In the context of a hygienic metaprogramming system, tailoring rules can even harmoniously compose with one another.

Type tailoring has emerged as a theme across several languages and metaprogramming systems, but never with direct support and rarely in the same shape twice. For example, both OCaml and Typed Racket enable forms of tailoring, but in quite different ways. This paper identifies key dimensions of type tailoring systems and tradeoffs along each dimension. It demonstrates the usefulness of tailoring with examples that cover sized vectors, database queries, and optional types. Finally, it outlines a vision for future research at the intersection of types and metaprogramming.

2012 ACM Subject Classification Software and its engineering → Extensible languages

Keywords and phrases Types, Metaprogramming, Macros, Partial Evaluation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.30

Category Brave New Ideas

Supplementary Material
Software and Data (Artifact): `https://doi.org/10.5281/zenodo.12726060` [107]

Funding This work was partially supported by DOE Office of Science Contract DE-SC0022252, XStack, ComPort, "Rigorous Testing Methods to Safeguard Software Porting" and by NSF grants SHF 1518844, CCF 2217154, and CCF/CSE 2030859 to the CRA for the CIFellows project. Felleisen's research was partially supported by several NSF grants (SHF 2007686, 2116372, 2315884).

Acknowledgements Thanks to Sam Tobin-Hochstadt for inspiring tailoring in Typed Racket, to Mark Ericksen for teaching best practices of Phoenix Verified Routes, to Ryan Culpepper for syntax parse support, to Matthew Flatt for help with Rhombus annotations, and to Alex Knauth, Asumu Takikawa, Gabriel Scherer, Justin Slepak, Leif Andersen, Scott Wiersdorf, and Zeina Migeed for comments on early drafts.

1 Type Tailoring Helps Programmers

Every typed language should come with a *type tailoring* toolkit to let programmers systematically rewrite code before it reaches the typechecker. Tailoring is essential for keeping up with libraries, domain specific languages, and even built-in embedded languages. Consider regular expressions, which are often embedded in strings. The following example builds a

regular expression that matches and extracts a two-digit number from a username. Without tailoring, the Typed Racket typechecker rejects this program:

Language: Typed Racket

```
(define dig-pat "[0-9]")
(define two-digs (string-append dig-pat dig-pat))

(define (user-idnum (username : String)) : Number
  (define full-pat (string-append "(" two-digs ")"))
  (define m (regexp-match full-pat username))
  (if m
      (string->number (second m))
      (error "bad username")))

(user-idnum "dent42")
```

Without Tailoring	With Tailoring
Type error: `second` argument: `(Listof (Option String))` expected result: `String`	Success: 42

Programmers fluent in regular expressions know that a call to `user-idnum` returns a number when `regexp-match` succeeds; otherwise, `user-idnum` raises an error. A standard type system knows much less because it ignores the values of strings: it does not know that the first capture group `(second m)` must exist when the match succeeds, and it does not know that the call to `string->number` must also succeed. To resolve these issues, the typechecker requires explicit checks and casts. Type tailoring can insert sufficient casts automatically by analyzing the regular expression string and transforming the program, thereby convincing the typechecker that the code is safe. For programmers, the net result is that the code above just works—without the clutter of casts, and without the need to migrate to an alternative regular expression syntax [57, 94, 106].

Type tailoring improves type analysis by observing and propagating static information. First, a tailoring for regular expressions observes the structure of the pattern string and finds that it has balanced parentheses that enclose a two-digit pattern:

```
(define full-pat
  (string-append "(" two-digs ")"))
```

Tailoring sees: `full-pat` is a literal: `"([0-9][0-9])"`.

If the string were not available for static analysis there would be nothing to observe:

```
(define hidden-pat (read-line))
```

Tailoring sees: `hidden-pat` is a string.

Second, tailoring propagates information about the pattern string from where it is defined to where it is used. Since this example is implemented in Typed Racket, it uses Racket syntax properties to store and retrieve metadata; other languages use similar mechanisms (Section 3). At the `regexp-match` call, this static information implies that the result of a successful match must be a specific list value:

```
(regexp-match full-pat username)
```

Tailoring sees: since `full-pat` has one capture group, the match returns either `#false` or a list of two strings, the second of which consists of two digits.

Third, tailoring elaborates (rewrites) the `regexp-match` call to cast a successful match result, thereby communicating the structure of the result value to the typechecker. Similarly, tailoring can elaborate the access `(second m)` to a faster, unsafe memory access because it is sure to be in bounds when the match succeeds:

```
(if m (second m) ...)
  ⇝ (if m (unsafe-ref m 1) ...)
```

Tailoring sees: since `m` is a list in this branch, it has two strings.

Information about the extracted string flows into a tailoring for `string->number` and justifies a final cast to convince the typechecker that the result must be number.

```
(string->number •)
  ⇝ (cast (string->number •) Number)
```

Tailoring sees: the cast cannot fail because • evaluates to a two-digit string.

Stepping back from this example, the overall message is that incorporating a bit of partial evaluation, flow analysis, and metaprogramming into the front end of a conventional typechecker is an effective way to support domain-specific typing for embedded languages. Type tailorings can compose with one another and can enhance an entire module with a few changes to its preamble (e.g. by importing a tailored regular expression library) rather than whole-program edits. There is of course a risk that arbitrary tailorings can perform unexpected or unsafe elaborations, but we show in Section 5 how the authors of tailorings can mitigate these concerns by appealing to baseline program behavior and static information.

Contributions

Type tailoring has appeared in many contexts, but never as an officially-supported language feature. This paper analyzes the spectrum of type tailoring across languages and libraries, identifies the linguistic features that make tailoring work, and establishes a framework for future research to push the boundaries of *end-user* programmable type elaboration. Concretely, the paper makes the following contributions:

- it proposes *type tailoring* as an overarching concept in user-level metaprogramming that should be recognized and more-widely adopted;
- it demonstrates the usefulness of tailoring with a variety of examples using Typed Racket, Rhombus, Julia, and Elixir (Section 2);
- it analyzes tailoring systems along six technical dimensions (Section 3), thereby revealing three notable points in the language design space (Section 4); and
- it provides a recipe for reasoning about the behavior of tailorings and establishing the validity of their transformations (Section 5).

The paper concludes with related work (Section 6), future work (Section 7), and takeaways (Section 8).

2 Tailoring in Action

To illustrate the variety of type tailoring, this section presents five examples in four languages: a tailoring framework and type programming in Typed Racket [101], dynamic typing in Static Rhombus [33], sized arrays in Julia [7], and statically-checked web routes in Elixir [28]. Each example contributes a unique perspective to showcase the range of type tailoring applications. Typed Racket supports a family of related tailorings, Rhombus weakens types instead of strengthening them, Julia achieves order-of-magnitude performance improvements, and Elixir shows how tailoring can benefit an untyped language.

2.1 Refining Data in Typed Racket

Typed Racket enables type tailoring by exposing key pieces of Racket's macro API and by typechecking code after macro expansion [21, 103]. Macros can thus inspect and transform

code to manipulate what the typechecker sees. The `trivial` library [40] uses this API to tailor a variety of domains from `printf` strings to SQL queries and beyond.

Format Strings

For `printf`, tailoring uncovers static information from escape characters in format strings. This tailoring stores information in a dictionary, mapping the key `fmt-args` to the expected types of the remaining arguments of `printf`. (Through the use of a key-value store, multiple tailorings can work together, as we will see later in this section.)

Typed Racket

```
(require trivial)

(define fmt1 "hello ~a\n")
(define fmt2 "int to bin: ~b\n")
```

:: { *fmt-args* : `[any]` }
:: { *fmt-args* : `[Integer]` }

Calls to `printf` statically check whether their first argument has format information. When this information is present, the tailoring checks the number and type of other arguments and raises an error at compile time if there is a mismatch. Without tailoring, such checks do not happen until runtime:

Typed Racket

```
(printf fmt1 "world")
(printf fmt1 "john" "hancock")
(printf fmt2 "NaN")
```

Without Tailoring	With Tailoring
Runtime errors: `printf` – `[fmt1]` expected 1 argument, given 2 – `[fmt2]` expected an integer, given something else	Tailoring errors: – `[fmt1]` expected 1 argument, given 2 – `[fmt2]` expected `Integer`, given `String`

Query Strings

For SQL queries, two sources of information come together to provide static checks via tailoring: database schemas and query strings. Programmers must write the schemas as type annotations in a notation specified by the SQL tailoring. Query strings use conventional SQL syntax to access the database. The tailoring parses query strings to reveal type constraints.

In the following example, the schema argument states that the database has one table named `Cats` with three columns for an identifier, pet name, and breed. Tailoring elaborates the `query-row` call to validate argument types. Without tailoring, the database executes the nonsensical query and returns an empty result:

Typed Racket

```
(define db
  (sqlite3-connect #:user "user"
                   #:database "Pets"
                   #:schema [Cats
                             [(id : Integer)
                              (name : String)
                              (breed : String)]]))
(query-row db
           "SELECT breed FROM Cats
            WHERE name = ?"
           69105)
```

Without Tailoring	With Tailoring
Runtime error: `query-row` query returned zero rows	Tailoring error: expected `String`, given `Integer`

This tailoring additionally propagates information about the result of a query. When a query selects only the `breed` column, the result is a vector with only one string value:

```
(query-row db                                    Typed Racket
           "SELECT breed FROM Cats
            WHERE name = ?"
           "mittens")
```
:: { *type* : `(Vector String)` }

While the `regexp-match` example from Section 1 and the `printf` example from this section improve code with no effort from users, the SQL tailoring cannot act without a schema as input. But, since the tailoring works through surface syntax, it can get this input in an idiomatic way without being constrained by the typechecker or host language. In particular, tailoring adds support for the `#:schema` argument by parsing the schema and elaborating to a plain `sqlite3-connect` call with only two keyword arguments.

Cooperating Tailorings

Static information embedded in strings can be useful in domains beyond `printf`, database queries, and regular expressions. By storing domain-specific information in a dictionary and using uniquely generated keys (Section 3.4), different tailorings can annotate the same value. For example, the following string has at least three interesting properties:

```
(define str                                      Typed Racket
  "(SELECT breed FROM Cats)")
```
:: { *rx-groups* : 1, *db* : `[SELECT (breed) Cats]`, *string-len* : 24 }

Of course, strings are not the only data structure that get repurposed in domain-specific ways. Vectors, lists, functions, and numbers also benefit from tailoring:

```
(define buffer                                   Typed Racket
  (make-vector (expt 2 5)))
```
:: { *vector-len* : 32 }

```
(define (swap ab)
  (list (second ab) (first ab)))
```
:: { *fn-arity* : 1 }

```
(define pairs '((1 2) (3 4)))
(map swap pairs)
```
:: { *list-len* : 2 }
:: { *list-len* : 2 }

In all cases, there is a general recipe at hand:

- Static information originates in surface syntax, such as the characters in a string literal or the shape of a function declaration. Information can also come from an external source, such as a database schema or online API specification.
- When static information is present, type tailoring attaches it to variables and propagates it through operations such as `map` and `regexp-match`.
- Tailoring elaborates surface syntax to code that the host typechecker can understand.

Defining a new tailoring requires three steps. First, define a unique key (e.g. via `gensym`). Second, create tailored variants of constructors that identify and attach static information. Third, create wrapper macros (i.e., compile-time functions) around various operations to leverage static information when it is present and otherwise preserve the default behavior.

Evaluation: Typing Regular Expressions

Regular expression tailoring comes with immediate benefits for typed code because, by default, programmers must use casts to guard against match failures even when such failures obviously cannot occur. In Typed Racket, the need for casts arises from the conservative type of `regexp-match` results, which says that all match results are either `#false` or lists with

one string element and an unknown number of additional elements that are either strings or `#false`. This type is always correct but usually too imprecise to be useful:

Typed Racket

```
(regexp-match full-pat username)
: (Option (Pairof String (Listof (Option String))))
```

Searching the Racket 6.5 distribution and code on its package server revealed 160 files using `regexp-match` with capture groups. Migrating the files to use type tailoring obviated the need for casts in 116 originally-untyped files and 6 typed files. These improvements resolved a total of 329 false type errors (that Typed Racket would have reported) across 93 % of all `regexp-match` occurrences in the dataset [107].

Only 38 files were not improved by tailoring. Most of these files (20 of 38) extracted a capture group, but did not depend on the result being a string. The others either used non-constant pattern strings (5 files), used helper functions to assemble patterns (9 files), or used patterns with groups that may indeed fail to capture—such as `"(a)|(b)"` (4 files).

Evaluation: Predicting Vector Bounds

Racket library code occasionally employs fixed-size vectors. For example, the built-in `gzip` implementation declares vector constants to implement a Huffman tree. When these vectors get accessed with a statically-known index, tailoring can refine code to skip the bounds check.

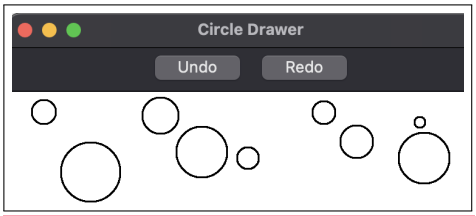
Across the Racket core distribution and packages, we found 88 files using vector constants. Tailoring eliminated bounds checks in 11 files. This number is low, but within these few files, tailoring improved 104 bounds checks in total. Most of these (80 of 104) appear in a Parcheesi implementation that uses a macro to generate code that accesses valid locations; this example shows that tailorings are robust even when other metaprogramming is present.

2.2 Elaborating Types in Typed Racket

Types themselves can benefit from tailoring. Since types are mere syntax before the type-checker gives them meaning, tailoring can elaborate declarative syntax into fine-tuned types.

Concise GUI Subtypes

Felleisen's implementation [29] of the 7GUI benchmark [52, 53] uses tailoring to simplify class type declarations. In Typed Racket, subclasses must declare types for all inherited methods and fields. While this requirement means that a subclass may refine the types of methods defined in its superclass, it also imposes a significant burden on programmers. In 7GUI, subclasses of the `Canvas%` type would normally have to spell out the types of thirteen methods. Type tailoring lets programmers specify just the differences from the parent class:



TR

```
(define-type-canvas Circle-Canvas%
  #:minus-init (paint-callback)
  (unlock (-> Void))
  (draw-circles
    (-> (Optional Circle)
        (Optional (Listof Circle))
        Void)))
```

Without Tailoring

13 method declarations (not shown)

With Tailoring

2 method declarations, 1 subtraction

The tailored 7GUI specification for a circle-drawing canvas simply gives the name of one constructor input to remove (`paint-callback`) and the types for two methods to add: an

`unlock` method to freeze the canvas (`lock` is private) and a method to draw circles on the canvas. Without tailoring, this specification would require twelve redundant names and types from the parent class.

Functional-Style Object Types

The `zombie` program from the GTP benchmark suite [41] presents an example of types tailored for readability. The original, untyped program uses functions to mimic message-passing objects [105]. Equipping such a program with types is challenging. For example, the function `new-zombie` takes a coordinate pair (`Posn`) and returns another function (`Zombie`) from a symbol to a method representation. These method representations are pairs that combine a label and function. In the code below, there are two methods that have different types:

Typed Racket

```
(define (new-zombie p) ;; Posn -> Zombie
 (λ (msg)
  (case msg
   [(can-grab?)
    (cons 'can-grab? ;; Method 1: Posn -> Bool
          (λ (q) (<= ((posn-dist p) q) *grab-radius*)))]
   [(move-to)
    (cons 'move-to   ;; Method 2: Posn -> Zombie
          (λ (q) (new-zombie ((posn-move-to p) q *speed*))))]
   [else
    (error "unknown message")])))
```

Although Typed Racket can express the overloaded return type for a `Zombie` object (via singleton types for labels [43]), writing such types requires intimate knowledge of the encoding. With tailoring, the types can essentially match Typed Racket's object type syntax:

Typed Racket

```
(define-type Zombie
  (-> Symbol
    (U (Pair 'can-grab? (-> Posn Bool))
       (Pair 'move-to (-> Posn Zombie)))))
```

TR

```
(define-obj-ty Zombie
  [can-grab? (-> Posn Bool)]
  [move-to (-> Posn Zombie)])
```

Without Tailoring

Encoding with pair and union types

With Tailoring

Domain-specific representation

Type Expanders

The type expanders library [93] can build types such as `Zombie` and `Circle-Canvas%` *within* another type, without the need to declare a top level definition via `define-type`. The following example, from the library documentation, presents a type expander `HomogeneousList` that uses an integer literal to expand to a `List` type:

Typed Racket

```
(: five-strings (-> String (HomogeneousList String 5)))
(define (five-strings x)
  (list x "a" "b" "c" "d"))
```

Without Tailoring

```
(List String String String String String)
```

With Tailoring

```
(HomogeneousList String 5)
```

Type expanders supports a wide range of additional tailorings. These include type abstraction (Λ) and local definitions in types (`Let`).

2.3 Relaxing Types in Rhombus

Rhombus enables tailoring in the same manner as Typed Racket by inheriting Racket's metaprogramming tools [33]. To illustrate, we equip the static variant of Rhombus (akin to strict JavaScript [25]) with a dynamic type (`Dyn`) in the spirit of optional and gradual typing [91, 99, 100]. Gradually typed languages are often defined by elaboration into a typed language with casts, making them a natural application for tailoring.

Rhombus comes with an annotation language that can, among other things, statically resolve method calls. For example, a function whose argument has the `List` annotation knows where to find the appropriate `length` method for this argument:

Static Rhombus

```
fun len_plus_one(l :: List):
  l.length() + 1
```

Static Rhombus requires annotations for the receivers of all methods calls, all list and map lookups, and similar operations [79]. It uses these annotations to guard against errors and to compile optimized code. Getting the annotations right can become a burden, as the long history of gradual typing attests [39, 67, 92, 102]. It would be useful to selectively disable the requirement, but by default Rhombus provides only a coarse-grained solution via a keyword `use_dynamic` that disables annotation requirements within an entire block of code.

The `Dyn` tailoring is an annotation that selectively disables static checks for an individual variable without losing guarantees in other parts of the same code block. This can be useful, for example, to implement a dynamic equality function for lists:

Static Rhombus

```
fun list_equals(l1 :: List.of(Dyn), l2 :: List.of(Dyn)):
  def len = l1.length()
  len == l2.length()
    && for all (i: 0..len):
      l1[i].equals(l2[i])
```

Without Tailoring	With Tailoring
Compile error: `l1[i].equals` no such method based on static information	Success

Static Rhombus accepts this code and resolves the call to `.equals()` at runtime. Other calls, such as `l1.length()` and the `for`-comprehension iterator, resolve statically.

Evaluation: Using `Dyn` in Shplait

Shplait is a typed, ML-like language developed for teaching and implemented in Rhombus [33]. It is one of the largest Rhombus programs to date. Within the Shplait codebase, there are 84 type annotations that appear in function, variable, and class definitions. These annotations appear in 21 of the 46 core Shplait files. Replacing these annotations with `Dyn` does not raise any compilation errors. The only difficulties that arose were due to import clashes with operations that `Dyn` overrides (such as `++`), which were straightforward to resolve by importing `Dyn` with a prefix.

2.4 Static Arrays in Julia

Array accesses in Julia incur a runtime bounds check by default. This check can become a significant and unnecessary cost in scientific code. For example, the coordinates for bodies in an N-body simulation might be stored in fixed-length arrays that get accessed in a highly repetitive pattern. A bounds check on each access would quickly incur a significant and unnecessary performance cost.

The StaticArrays package implements a form of type tailoring that elaborates normal-looking array code into fixed-size tuples [96]. To declare a static array, programmers wrap a normal array declaration in the `@SVector` macro. The package supports several kinds of declarations, including array comprehensions:

```
vec1 = @SVector [1]
vec3 = @SVector zeros(3)
vec9 = @SVector [i^2 for i = 1:9]
```
Julia

$$:: \{\ \mathit{vector\text{-}len} : 1\ \}$$
$$:: \{\ \mathit{vector\text{-}len} : 3\ \}$$
$$:: \{\ \mathit{vector\text{-}len} : 9\ \}$$

In addition to array constructors and references, StaticArrays tailors a variety of linear algebra operations to use size information. Matrix multiplication, transposition, and reshaping can all propagate sizes. Eigenvalue decomposition uses a fast algorithm for small matrices:

```
m3 = @SMatrix randn(3,3)
eigen(m3).values
```
Julia

$$:: \{\ \mathit{matrix\text{-}shape} : (3, 3)\ \}$$
$$:: \{\ \mathit{vector\text{-}len} : 3\ \}$$

Elaborating arrays to tuples is impractical for large arrays in Julia; the StaticArrays documentation recommends 100 elements as a rule-of-thumb upper bound. Nevertheless, StaticArrays is an important part of the Julia ecosystem. As of January 2024, it has over 800 direct dependents and 3,000 indirect dependents (30 % of the 10,292 packages on JuliaHub). One of its clients is the popular OrdinaryDiffEq package [88], which helps explain the large number of indirect dependents.

Evaluation: Fast Matrix Rotations

The documentation for StaticArrays reports order-of-magnitude speedups on a variety of linear algebra microbenchmarks using 3×3 matrices [96]. Examples include a 5.9 x speedup for matrix multiplication, a 113 x speedup for determinant computation, and a 8.8 x speedup for Cholesky decomposition. We built our own microbenchmark that rotates a vector by a 10×10 matrix one hundred million times. The results in Table 1 show a 10 x speedup and a dramatic decrease in memory use thanks to inlining and predictable array layout. These numbers are the average after two hours of sampling with `BenchmarkTools.jl` in Julia 1.8 on a single-user Linux machine with 4 physical i7-4790 3.60GHz cores and 16GB RAM.

Table 1 StaticArrays yields a 10x speedup on a synthetic matrix rotation benchmark

	mean (stddev)	Memory Use	GC % (stddev)	samples
Without Tailoring: Arrays	10.7 s (8.3 ms)	13 GB	1.7 % (0.07 %)	671
With Tailoring: StaticArrays	1.5 s (8.9 µs)	0 B	0 % (0 %)	4856

2.5 Verified Web Routes in Elixir

The Phoenix web framework [75] uses Elixir's macro system to validate web routes. Programmers declare routes and corresponding handlers in a dedicated module. Phoenix leverages information from the route module in a three-step validation process: first, it collects route references that marked by a certain macro (`~p`); second, it elaborates these marked references into plain strings; and third, it checks that each reference has a matching handler.

Route references commonly appear in page templates. For example, the following template code block contains the references `/users/register` and `/users/login`. Suppose that the second route has a typo, and the correct path is `/users/log_in` with an underscore. Without tailoring,

routes are mere strings; a typo in a route name is not a problem until a user requests the page and gets a 404 error. With tailoring, a static check catches the error immediately.

Elixir

```
<p>
  <%= link "Register", to: ~p"/users/register" %>
  <%= link "Log in", to: ~p"/users/login" %>
</p>
```

Without Tailoring	With Tailoring
Possible 404 at runtime	Tailoring error: no route path matches `/users/login`

Thus, even a dynamically typed language such as Elixir can benefit from domain-specific static checks during the elaboration of source code to baseline Elixir. Prior work in Scheme illustrates the same point in several other domains [44], though at a much smaller scale than Phoenix. Ruby on Rails [6] and Haskell [65] have their own methods of static route validation; this is a common issue that tailoring helps to solve.

Evaluation: Adoption Data

Phoenix introduced verified routes in February 2023 [64]. They are an optional feature for existing projects, while for new projects Phoenix generates code with verified routes by default. As of January 2024, over 1,800 Elixir files on GitHub are using verified routes [36].

3 Dimensions of Tailoring Systems

Many languages and libraries support a form of type tailoring. Examples include the macro systems in Clojure [18], Scala 2 [9], Scala 3 [86], and Rust [85]; elaborators in Idris 1 [13]; Template Haskell [89]; OCaml PPX [69], MetaOCaml [51], and MacoCaml [109]; CompRDL [49]; and type providers in the style of F# [14, 74]. Despite differences in their specific aims and affordances, they all enable metaprogramming of the elaboration from surface code into typechecked code.

As a step toward an analysis of language support for tailoring and of relative strengths and weaknesses, this section introduces a framework of *technical dimensions* (inspired by prior work [38, 45]) for type tailoring. The dimensions fall into two groups: the first four describe metaprogramming features, and the last two describe contextual information that may be available to tailorings:

Metaprogramming Features

Metadata For tailorings to work together, they must have a way of sharing static information. Metaprogramming systems that can attach metadata directly to AST nodes enable this sharing in a direct way. (Compile-time state is an alternative.)

Binding How to handle bindings is a crucial aspect of information sharing. Information must be able to flow from a variable declaration to its use. Metaprogramming systems can discover these connections and expose them to tailorings.

Order Cooperating tailorings need a reliable and customizable order of expansion. One tailoring might depend on input from another, and it may wish to have a third tailoring analyze its output.

Hygiene To a first approximation, a hygienic metaprogramming system respects the lexical structure of code. Hygiene is important for tailorings to compose with one another and with user code. Users, for example, should not need to worry about whether the `f` in `f(x)` is a tailoring (e.g., a macro) or a normal function.

Table 2 Technical dimensions of tailoring systems

	Metaprogramming				Context	
System	Metadata	Binding	Order	Hygiene	Definitions	Types
Racket	●	●	●	●	●	×
Clojure	●	●	●	○	●	×
Elixir	●	×	●	●	●	×
Julia	○	×	●	○	●	×
Idris 1	×	○	○	×	●	●
Scala 3	×	×	●	●	●	●
Template Haskell	×	×	○	●	○	×
Type Providers	×	×	×	×	●	○
OCaml PPX	○	×	×	×	○	×
Rust	×	×	○	○	●	×

● Full support ○ Partial support × No support

Context Information

Definitions Tailorings benefit from local definitions and external data, such as a database, as sources of static information. Without definitions, tailorings are limited to local transformations such as refining calls to `printf` that apply a literal format string.

Types Type context is a dimension that has benefits and drawbacks. On one hand, if tailorings receive typechecked input then they can leverage the types and need not handle malformed syntax. On the other hand, types restrict the shape of domain-specific syntax to terms of the host language.

Table 2 presents an evaluation of representative tailoring systems along these technical dimensions. The rows are *not* an exhaustive list of tailoring systems but rather give an overview of distinct feature-sets. For example, there is no row for Rhombus because it has the same feature set as Racket. The table suggests two high-level takeaways:

1. A number of systems lack support along several dimensions. These represent trade-offs that realize some benefits at a modest effort. Section 4 examines three points in depth.
2. No system has full support along every dimension. Section 7 presents ideas for designing a full-featured system as future work.

The rest of this section explores the cells of Table 2 in detail. For each dimension, a subsection provides a detailed description of what it means and justifies the partial cells (○). The last subsection gives a concrete implementation of a tailoring for static vector references; this example shows how one tailoring benefits from several dimensions.

3.1 Metadata

Static information is metadata. Examples include the length of a string literal, the capture groups in a regular expression, and the schema of a database. Tailorings discover this metadata, and they need a way to disseminate it to reap the benefits. For example, discovering the length of a literal vector may be useful by itself, but it is more useful if length information can propagate through operations such as vector concatenation.

Attaching metadata to AST nodes is a direct way to propagate information. Any piece of syntax—whether it describes a value, an expression, or a definition—should support metadata. Furthermore, as the examples from Section 2.1 demonstrate, the metadata should be a key-value store that can hold data structures as values. Keys clarify the interpretation of data such as numbers, which, in the examples, represent lengths and regexp groupings. Structured values declaratively express format-string constraints and database schemas.

Racket [77], Clojure [19], Rhombus [80], and Elixir [27] support general key-value metadata on arbitrary nodes. Julia [47] and OCaml [69] support a limited form of metadata; only a specific type of AST node can hold metadata. These metadata nodes can, however, be inserted as siblings to other nodes in the syntax tree. To the best of our knowledge, the other systems in Table 2 have no direct support for metadata, though Idris has highly-expressive dependent types that can achieve similar goals.

3.2 Binding

Variable declarations call for a special kind of metadata that flows from a binding to its references. Consider a basic `let` expression that binds a format string to a variable `str`; a tailoring system should ensure that calls to `printf` within the body have access to format metadata:

Racket
```
(let ([str "age: ~a"])
  ... (printf str n) ;; need data here
  ... (lambda (str) (printf str n))) ;; but not here
```

In Racket and Rhombus, *rename transformers* flow data to references [78]. Clojure has libraries for similar functionality [17, 63]. None of the other systems support renaming in a programmatic manner, which means that the authors of tailorings need to manage propagation on their own—perhaps by handcrafting AST structures.

3.3 Order

Control over the order of elaboration makes it possible for tailorings to share their results with one another. For example, static-length vectors and constant folding are somewhat useful in their own right, but are more effective together:

Typed Racket
```
(define buf-size (* 4 4))          :: { int-value : 16 }
(define buffer (make-vector buf-size))   :: { vector-len : 16 }
(sub1 (vector-length buffer))      :: { int-value : 15 }
```

Unlike in a normal metaprogramming system, it is crucial that the order of elaboration matches the order of runtime evaluation. The tailoring for `make-vector` must happen after the tailoring for multiplication, and the tailoring for `sub1` must happen last of all because it depends on the first two results.

Clojure macros give control over ordering in a simple way through a `macroexpand` directive [68] inherited from Lisp [97] and Scheme [23]. Julia [48] and Elixir [27] provide a similar directive. This method is unhygienic, and may cause problems when macros depend on one another [31]. There are several alternative forms of sequencing that fall short of arbitrary ordering. Elixir provides compile-time hooks to register code that runs before and after a module compiles; these hooks let Phoenix (Section 2.5) verify routes after registration. Template Haskell allows stacks of tailorings, but programmers must manage them explicitly by wrapping each piece of syntax in a suitable number of template quotes [89]; the Haskell

type system does help to manage the layers. Idris elaborators require similar management [13]. Rust macros can expand to other macros that have been defined in a separate crate [85]. Type providers in F# cannot expand to one another [74]. OCaml PPX recommends that users do not rely on the order of expansion [69].

3.4 Hygiene

A hygienic metaprogramming system enables composable tailorings. Macro hygiene ensures that compile-time transformations respect the binding structure of the code they manipulate. A classic illustration is an `or` macro that introduces a temporary variable:

Common Lisp
```
(defmacro or (a b)
  `(let ((tmp ,a))
     (if tmp tmp ,b)))
```

If a call to this macro simply replaces code, the `tmp` variable can shadow a binding and produce the wrong result:

```
;; before expansion
(let ((tmp 42))
  (or nil tmp))

;; expected result: 42
```

```
;; after unhygienic expansion
(let ((tmp 42))
  (let ((tmp nil))
    (if tmp tmp tmp)))
;; actual result: nil
```

A second hygiene issue concerns references from macro definitions to functions. For example, the SQL tailoring from Section 2.1 relies on helper functions to analyze strings. If the helpers' names were to get shadowed at the macro use-site—as is the case with unhygienic systems—the tailoring would crash or produces flawed results.

A third related issue is that tailorings ought to work as a drop-in replacement in user code. Code that calls a standard function such as `make-vector` should work with a tailored variant instead, without the programmer needing to annotate the call site as a macro call rather than a function call. Since functions are typically first-class values, this means that macros must work hygienically in first-class use-sites.

Lastly, the keys used to label static information need a form of hygiene. If two tailorings inadvertently choose the same key, they may attach conflicting information to a value. Tailoring systems must provide a facility to generate unique keys—such as `gensym` in Julia and other languages—to prevent clashes.

Racket [32], Rhombus [33], Elixir [27], Scala 3 [87], Template Haskell [89] all support macro hygiene. Julia is hygienic for simple macros, but in complex macros programmers must manually rename variables to avoid issues [48]. Rust's support for hygienic macros is currently experimental [50, 85]. F# type providers, OCaml PPX [69], Idris [13], and Scala 2 [9] provide no support for hygienic macros.

3.5 Definitions in Context

There are two aspects of definitions that relate to tailoring. The first and most important is access to external data, whether it be a relational database (Section 2.1), a manifest of web routes (Section 2.5), or a JSON endpoint [74]. To add two more examples to the mix, CompRDL uses domain-specific knowledge of Ruby on Rails and database schemas to achieve dependent types for table functions [49], and the Rust SQLx library checks the well-formedness of query strings [58]. Both leverage external data to analyze code without asking programmers to change their idiomatic code (conventional Rails in Ruby, raw SQL in Rust). Every tailoring system in Table 2 provides access to external data.

The second aspect of definitions is access to local variables and helper functions. As mentioned in Section 3.4, a tailoring that parses query strings benefits from access to helper functions. (Without access, the tailoring must duplicate code in its definition, which then increases the size of the object code.) Most systems provide access to local definitions, though in the context of unhygienic systems this must be done with care. Template Haskell code can access local definitions but is subject to Haskell's disciplined use of side effects [89]. OCaml PPX runs macros in isolation, so they cannot use compile-time definitions [69].

3.6 Types in Context

In Scala 3, the typechecker runs before tailorings do. Types provide extra context to tailoring and detect certain malformed input. However, types also put restrictions on inputs. In the Squid DSL for Scala, programmers must hide code in strings to get it past the typechecker [73]. The following example is from the Squid documentation [72]:

```
val powCode = code"${(x: Variable[Double]) => mkPow(code"$x", Const(n))}"
```

Idris distinguishes between typed and raw terms during elaboration [13]. All terms must eventually pass the typechecker, but elaboration can manipulate both sorts of terms. The other systems in Table 2 do not typecheck their input, though type providers can effectively rely on types because their inputs are restricted to literal constants. There are, however, several metaprogramming systems in the literature that do typecheck their inputs. Examples include SoundX (which furthermore guarantees well-typed transformations) [61], Wyvern [70], Dependent ML [108], and Scala 2 [9].

3.7 Essential Non-Dimensions

Table 2 focuses on elements that are significant for tailoring but lack full adoption. As such, it omits basic features that enable type tailoring. The following are requirements rather than dimensions, but they are nevertheless important for language designers to know about:

Typechecking After Elaboration Regardless of whether or not typechecking happens before elaboration, it must happen afterward to check the results of user-defined tailorings.

Elaboration-Time Computation Tailorings need infrastructure, such as procedural macros, to perform non-trivial computations. By contrast, pattern-based macros (which unpack the syntax of their call-site and rearrange it [55]) cannot even read from an external file.

AST Datatype Without an AST datatype, tailorings are limited to using a token stream as input and output (e.g., in Rust [85]). Token streams cannot carry metadata (though it might be stored off to the side) and must be parsed to find their binding structure.

3.8 Example Tailoring Implementation

Let us show how the dimensions work together with an example that tailors vector references in Racket. This demo replaces `vector-ref` with either a fast `unsafe-vector-ref` or an error when it finds both a literal vector and a literal index; otherwise, it leaves the reference as is:

```
(vector-ref (vector 5 2 8) 1)  ⇝  (unsafe-vector-ref (vector 5 2 8) 1)
(vector-ref (vector 4 9 1) 4)  ⇝  error: out-of-bounds
(vector-ref (read-vec) 9)      ⇝  (vector-ref (read-vec) 9)
```

See the documentation of the `trivial` library [40] for a full-featured implementation that works for identifiers as well as data literals.

The six code blocks below define the `vector-ref` tailoring. When defining the tailoring, we use the name `tailored-vector-ref` to avoid shadowing the base `vector-ref` function. On export, we rename this tailoring to `vector-ref` so clients of this library can use the tailored version as a drop-in replacement in their code:

Racket, 1/6
```
(provide
  (rename-out [tailored-vector-ref vector-ref]))
```

The tailoring module imports `unsafe-vector-ref` for use in expanded code, and three other libraries to define the tailoring:

Racket, 2/6
```
(require
  (only-in racket/unsafe/ops unsafe-vector-ref)
  (for-syntax ;; import these things for macros
    racket/base syntax/parse (only-in "tailoring-api.rkt" ⇝ ϕ V I)))
```

The helper module `tailoring-api.rkt` is a small wrapper over Racket's metaprogramming facilities. Specifically, it provides a bridge for the following dimensions:

Order of expansion ($\rightsquigarrow$) The syntax class $\rightsquigarrow$ triggers macro expansion on a subexpression, allowing the tailoring to discover static information.

Metadata (ϕ) The function ϕ uses Racket syntax properties to store and retrieve static information using domain-specific keys.

Hygiene (V,I) The keys V and I are unique keys (gensyms) for vector length and integer value information. (Under the hood, `tailoring-api.rkt` registers such information when it encounters relevant data literals during expansion.) Uniqueness means that other tailorings cannot accidentally use the same names and cause a collision; however, it does not stop a malicious tailoring from writing bad information using the keys.

For details on `tailoring-api.rkt`, refer to the artifact for this paper.

The tailoring itself is a macro so that it can statically rewrite source code. First, it parses its input syntax object (`stx`) to extract and expand two subexpressions:

Racket, 3/6
```
(define-syntax (tailored-vector-ref stx)
  (syntax-parse stx
   [(_ e1:⇝ e2:⇝)
```

The expanded form of subexpression `e1` is available as `e1.`$\rightsquigarrow$, and similarly for `e2`. The tailoring checks whether these expanded expressions have the static information that it needs; specifically, it needs a vector length (key: V) and an integer value (key: I):

Racket, 4/6
```
    #:do [(define n (ϕ (syntax e1.⇝ V)))
          (define i (ϕ (syntax e2.⇝ I)))]
    #:when (and (integer? n) (integer? i))
```

If the information is present, the tailoring checks whether the index is in bounds and expands to code that either performs a fast vector reference or raises an exception:

Racket, 5/6
```
    (if (and (<= 0 i) (< i n))
        (syntax (unsafe-vector-ref e1.⇝ e2.⇝))
        (syntax (error 'Index-Exn)))]
```

Otherwise, the default behavior is whatever Racket's un-tailored `vector-ref` does:

Racket, 6/6
```
   [(_ e1:⇝ e2:⇝)
    (syntax (vector-ref e1.⇝ e2.⇝))]))
```

Figure 1 Three levels of support for type tailoring

With that, the tailoring is complete. A Racket program can import this tailoring to replace the default `vector-ref`:

```
(vector-ref (vector 5 2 8) 1)
```
Racket

Without Tailoring	With Tailoring
No change; use checked lookup	Use unsafe, fast lookup
`(vector-ref ...)`	`(unsafe-vector-ref ...)`

In summary, tailoring `vector-ref` relies on a specific *order of expansion* to receive *metadata* about subexpressions, it extracts that metadata in a reliable way that composes with other tailorings using macro *hygiene*, and it relies on several *definitions* from the Racket standard library such as `<=` to compare numbers and Racket's `vector-ref` to provide a default behavior. Thus, four technical dimensions come together in this one example. Both the tailoring and its helper module are defined in user code; they require no changes to the Racket language to seamlessly improve client code.

4 Design Space Reflections

Tailoring systems come in wide variety in the literature, and yet they all enable at least some similarly-useful applications. For example, OCaml PPX achieves a pinch of dependent typing, F# type providers give an early warning when a web service changes its API, and Julia StaticArrays can propagate array dimensions. None of these systems can propagate metadata through binding forms, but their implementations are simpler than those that do. This tension suggests that there are points in the design space of tailorings that offer compelling tradeoffs between implementation complexity and useful capabilities.

In all, there are three important levels of tailoring support: (1) *local tailorings* that can use external data to generate types, (2) *cooperating tailorings* that share metadata and run in a customizable order, and (3) *binding-aware tailorings* that can manage an environment of static information. Figure 1 summarizes these levels both in terms of their required metaprogramming features and in terms of the tailorings these features enable. The bottom of the figure lists type information as a notable but orthogonal direction. Scala and Idris are the only systems that provide type information to macros, though it is unclear whether type information advances the frontier of tailoring systems.

4.1 Level 1: Local Tailorings

The first level of support for type tailoring enables local transformations that can query external sources of information. The PPX preprocessor in OCaml, SQLx in Rust, and type providers in F# all fall into this category. These systems can retrieve input from databases, websites, and/or constant literals to generate tailored code.

At this level, the main ingredient is support for compile-time computation: the preprocessor cannot be limited to simple pattern-and-template transformations. The preprocessor should also receive AST objects as input and it must be able to inspect these objects, query external sources, and have its output validated by the typechecker (compare to Section 3.7). Weakening any of these ingredients makes writing tailorings a challenge for language end-users, who do not have access to compiler internals. Without access to the AST, for instance, users must parse the input token stream before they can manipulate it in a meaningful way.

4.2 Level 2: Cooperating Tailorings

The second level of support allows tailorings to work together in basic ways. For example, `Dyn` in Rhombus uses metadata to tell operations how to handle an expression, StaticArrays in Julia lifts array structure into types, and Phoenix in Elixir collects web routes in a first pass before validating the routes in a second pass. All three require tools for sharing static information and controlling the order of elaboration.

Hygiene is crucial to enable sharing without bugs. In systems like Julia with partial support for hygiene, the authors of tailorings must fill the gap manually, for instance, by calling `gensym` to create fresh names.

Order in elaboration can come about in two ways. The direct way is to allow tailorings to expand to other tailorings. In this setting, the elaborator (macro expander) must continually process AST nodes until no tailoring uses remain. The indirect way, exemplified by Phoenix and CompRDL, is to use hooks to register tailorings that should happen at a certain point. In Phoenix, these points are just before and just after the compilation of a module.

Lastly, cooperating tailorings need a way to share information. An API for attaching metadata to AST nodes is the straightforward solution. Keeping information off to the side in metadata nodes (as in Julia) or in mutable structures is an alternative.

4.3 Level 3: Binding-Aware Tailorings

The third level of support is to equip the metaprogramming system with binding information. This level also allows fine-grained control over when the tailorings in a piece of syntax elaborate. Typed Racket requires these ingredients for its lightweight flow analysis, which lets tailorings flow through variable declarations and standard operations:

```
(let ([greeting "hello ~s"])
  (printf greeting "world"))
```
⇝
```
(printf "hello ~s" "world")
```

```
(make-vector (+ 40 2))
```
⇝
```
(make-vector 42)
```

To deal with variables, tailorings need access to the binding structure of code. To ensure that subterms elaborate before the outer term, tailorings need control over the order of expansion. Among the languages in Section 3, only Racket and Clojure give full control over binding structure. Manual macro expansion is more common, with support from Elixir [27], Julia [48], and Idris [13]. Only Racket and Rhombus provide both in a hygienic way.

5 How to Reason About Tailorings

Programmers need an easy-to-use guide for the design of correct tailorings because tailorings can rewrite source code arbitrarily. Tailorings come with a degree of safety because the host language typechecks their output, but types alone do not prevent an incorrect rewrite from $\sqrt{2}$ to 42 or an unsafe rewrite that accesses out-of-bounds memory.

In general, tailorings accomplish two goals: they discover and propagate static information, and they elaborate source expressions to the host language. These goals motivate two correctness requirements. The first requirement is *prediction soundness*. If tailoring attaches static information to an expression and the expression reduces to a value, then the static information must be a correct description of the value. For example, the key-value pair `{string-len:4}` is correct for string values with exactly 4 characters. If tailoring propagates static information for a variable, then the prediction must hold for all values the variable might take on.

The second requirement, *compatibility*, pertains to the behavior of elaborated code. Based on the examples from Section 2, there are three ways that a source expression and its tailored variant may relate to one another. Tailorings can:

- *express new behaviors* by translating invalid source syntax into valid host code;
  Examples: SQL in Typed Racket (Section 2.1), `Dyn` in Rhombus (Section 2.3).
- *refine existing behaviors* by changing how, but not what, an expression computes; or
  Example: StaticArrays in Julia (Section 2.4).
- *predict errors* by identifying a mismatch in static information.
  Example: Verified Routes in Elixir (Section 2.5).

Tailorings that refine behavior or predict errors can use the untailored program as a source of truth. They should be compatible in the sense of computing equivalent values or rejecting the same programs. Tailorings that express new behavior generally require a fine-tuned correctness argument, but they may benefit from the idea of compatibility as well. For example, the Typed Racket regular expression example in Section 1 should be compatible with the baseline behavior of untyped Racket.

A Recipe

Showing that a tailoring system is correct calls for a two-part effort. First, the designers of a *cooperating* tailoring system (Section 4) must show that any propagation rules or environment-management rules respect prediction soundness. The designers of a *binding-aware* system must show that metadata propagates correctly. The designers of a *local* system have no obligations at this stage.

Second, the authors of domain-specific tailorings have three tasks:

1. Confirm the prediction soundness of rules that infer static information from values. These may be straightforward, such as inferring a length from a literal vector, and they may be sophisticated, such as the F# algorithm for JSON shapes [74].
2. Categorize tailorings that elaborate expressions as either: expressing new behavior, refining existing behavior, or predicting errors.
3. Argue that the elaborations are acceptable. Elaborations should either be compatible with some baseline behavior or desirable in some other sense.

By way of example, the next three subsections present two domain-specific tailorings and one set of general propagation rules.

5.1 Express New Behavior: Variable-Arity Map

With arity information about functions, a language with simple function types (such as Haskell, but not Typed Racket [98]) can support a variable-arity `map` function by generating code for a fixed-arity `map`. In the examples below, $\texttt{map}_1$ expects exactly one list, while $\texttt{map}_2$ expects exactly two lists and applies the function to their elements in parallel:

```
(map add1 '(1 2 3))
  ⇝ (map₁ add1 '(1 2 3))
```

```
(map max '(1 2 3) '(4 5 6))
  ⇝ (map₂ max '(1 2 3) '(4 5 6))
```

When variable-arity `map` receives a function with unknown arity, or when the arity does not match the number of lists given, it raises an exception:

```
(map max '(1 2 3))
  ⇝ Exn: ‘max’ expects 2 arguments, but ‘map‘ got only 1 list
```

For the first piece of static information, we need a tailoring that discovers the arity of functions—for example, that that `max` takes 2 arguments:

```
(: max (-> Real Real Real))
(define (max a b)
  (if (>= a b) a b))
```

$:: \{\ \textit{fn-arity} : 2\ \}$

Prediction soundness comes from a function's arity being statically apparent. Likewise, predicting the number of list arguments is a simple matter of counting the arguments to `map`.

This tailoring is *expressing new behavior* when there is no variable-arity `map` in the source language. A reasonable baseline is to generalize the behavior of $\texttt{map}_1$, $\texttt{map}_2$, and so on in a compatible way. Assuming prediction soundness, `map` should elaborate to the correct $\texttt{map}_i$ or predict the error that $\texttt{map}_i$ would raise.

5.2 Refine Behavior and Predict Errors: Vector Bounds

When a tailoring expands to potentially-unsafe code, demonstrating soundness is crucial. Consider a tailoring that optimizes array references: when accesses (`vector-ref`) are known to be in bounds, it is safe to bypass the bounds check (`unsafe-ref`). However, if the index or the array length are not known statically, the tailoring falls back to a safe variant (`checked-ref`). This tailoring should always return the same result that a normal in-bounds access would, and should also optimize only when it is safe to do so:

```
(define my-vect (vector 1 2 3))

(vector-ref my-vect 1)           ⇝ (unsafe-ref my-vect 1)
(vector-ref (read-vector) 3)     ⇝ (checked-ref (read-vector) 3)
(vector-ref my-vect (read-int)) ⇝ (checked-ref my-vect (read-int))
```

If this tailoring detects an out-of-bounds access statically, it can predict an error at tailoring time instead of leaving it until runtime.

```
(vector-ref my-vect 42)          ⇝ Exn: Index '42' out of range
```

Information about vector sizes can come from two sources: static `vector` declarations and constructors such as `make-vector` that use statically-known sizes:

```
(vector 1 2 3)
```

$:: \{\ \textit{vector-len} : 3\ \}$

```
(make-vector 42 0)
```

$:: \{\ \textit{vector-len} : 42\ \}$

Prediction soundness follows from the semantics of these built-ins.

This tailoring is *refining existing behavior* by optimizing accesses that are known to be safe. It also *predicts errors* when it can statically discover an out-of-bounds access. With

prediction soundness in hand, behavioral soundness follows from comparing known lengths to known offsets.

5.3 Propagation and Substitution

Tailoring systems that propagate static information from variable definitions to references (i.e., *cooperating* systems in Section 4) must demonstrate *prediction soundness* for the forwarded information. Metadata must remain an accurate description for any runtime value the variable may take on. For instance, consider a vector bound to a variable `x`:

```
(let ([x (vector 1 2 3)])
  ... x ...)
```

`x` :: { *vector-len* : 3 }

In the body of the `let`, the identifier `x` should have the information `{vector-len:3}` attached to it. Subsequent tailorings inside the body of the `let` form should be able to take advantage of this information:

```
(let ([x ...])                ⇝        (let ([x ...])
  (vector-ref x 1))                        (unsafe-ref x 1))
```

At the same time, if `x` is shadowed by a different binding inside the body of the `let`, that same static information must not be attached to the new binding—transgressing lexical scoping would violate prediction soundness.

Aside from propagation through bindings, a *binding-aware* system can propagate information through a bottom-up tree traversal. For instance, `if` expressions can join the predictions from both branches:

```
(if (daylight-savings)
    (vector 1 2 3)
    (vector 4 5 6))
```

:: { *vector-len* : 3 }

In this example, both branches carry the same static information; but when the branches disagree, precise information cannot propagate upward:

```
(if (daylight-savings)
    (vector 1 2 3)
    (read-vector))
```

:: { *empty map* }

Exactly how to join pieces static information is context-dependent. For example, if both branches of an `if` are static vectors, but one is longer than the other, it might be sensible to propagate a length that is known to be safe. In other domains, defaulting to an empty set of properties might be the sensible thing to do.

6 Related Work

Type tailoring combines aspects of types, metaprogramming, and static analysis. In closely related work that inspired our Typed Racket tailorings, Herman and Meunier [44] show how a macro system can implement domain-specific static analyses for format strings, regular expressions, and database queries. They do not consider the interplay of domain-specific information and types. Ziggurat is a tailoring system for a C-like language [30]. It enables towers of language levels, each with a custom type system or flow analysis, and lets neighboring levels share information. Pluggable typecheckers add layers in a similar sense, though without direct support for sharing information across layers [8, 22, 66, 71]. Squid provides a framework for type-checked partial evaluation using the Scala macro system [73]. Scala LMS is another tailoring system specialized to partial evaluation that has enabled extensible optimizing

compilers for domains ranging from databases to linear algebra [81, 82, 83, 84]. The finally-tagless encoding of staged interpreters is a third technique for partial evaluation that can be implemented within a typed language such as ML [10]. While its applications are more limited than the tailorings in this paper, it avoids the need for a metaprogramming layer.

The ingredients of a full-featured tailoring systems are made possible by research from the Lisp family of languages [20, 21, 33, 34, 54, 55, 97]. In particular, we note the long line of research on hygiene [1, 15, 16, 32, 54, 76].

Tailoring achieves a modicum of dependent typing in the context of a simply-typed language and without the burden of formal proof. The `printf` and vector size tailorings are similar to work on dependent types [26]. Cayenne is related as a practical compromise with dependent types; its typechecker takes care of the proof burden, but may run indefinitely [3]. Dependent type systems are common targets for metaprogramming. Prior work includes the Lean 4 macro system [24], type-directed editing and elaborator reflection in Idris [13, 57], certified metaprogramming in Coq [2], and elaboration-time solvers in Agda [56, 59]. Extensible tactic languages such as Cur [11] and VeriML [95] are tailoring systems that turn concise proofs into elementary ones.

By contrast to metaprogramming systems in general, tailoring is limited to the elaboration of surface syntax to a typed host language. Closely-related systems include Turnstile [12] and Klister [5], which create whole typed languages; Haskell typechecker plugins [4, 42], which customize the type constraint solver; and Nx [104], which compiles Elixir-like source code to GPU kernels. On a similar note, the Haskell library Servant checks web APIs statically using typechecker extensions rather than metaprogramming [65].

7 Future Work

Developing improved support for tailoring is an ongoing challenge. In addition to the quest for a system that productively combines all technical dimensions from Table 2, the following are key areas of focus going forward:

Deeper Control Flow Analysis While the tailorings in this paper leverage some control flow analysis, this analysis is limited to local, forward propagation. No information flows through function calls, and join points (conditionals and loops) lose information to produce a conservative approximation. One way to recover precision is with annotations supported by runtime checks, though annotations put extra responsibility on programmers and checks introduce costs. Another way is to embed full [90] or demand-driven [35] control flow analysis in the tailoring system. Work on Turnstile is closely related, as it shows how macros can implement type analyses [11, 12].

Elaboration-Time Performance, How Much Metadata? The compile-time performance of tailorings has not been an issue for tailorings thus far, but it will become an issue if tailorings strive for whole-program or even whole-module analyses. Turnstile and k-CFA both suffer in this respect. A related issue is how much metadata to attach to AST nodes. Typed Racket tailoring (Section 2.1) inspects every data literal for every form of domain-specific information, which means that a string value can carry several kinds of information if it matches regular expression, `printf`, and SQL query syntax. Tracking information may, at some point, incur a noticeable compile-time cost.

How to Interleave Elaboration and Typechecking Most tailorings in this paper happen before typechecking, which leaves them free to accept DSL syntax but also forces them to accommodate ill-typed or even ill-formed programs. In Scala, typechecking happens before and after tailoring. Further research is needed to weigh the strengths of each approach on

concrete examples. On a related note, several features in Typed Racket including `match` and list comprehensions are implemented as macros and therefore get typechecked only after expansion. The typechecker struggles to reason about this post-expansion code. A source-level type analysis may be more straightforward.

Toward a Proof API Typechecking before expansion raises the question of how to share results with the host type system. The tailorings in this paper either produce simple code or use casts to share results. Other tailoring-adjacent systems, such as the units-of-measure Haskell plugin [42], merely assert results to the host language. One avenue for enhancement is to equip a standard typechecker with a tactic language (e.g., [37, 62]) to support proof objects. The System DE calculus implements a similar design for termination proofs [60]; it contains a sublanguage for constructing proof terms that the typechecker can accept and erase during compilation.

8 Discussion

Type tailoring is a lightweight way to grow [46] a type system by equipping it with additional expressiveness. Critically, tailorings are mere library code produced by and for ordinary users, rather than by compiler engineers. Additionally, tailorings cooperate with the existing type system and are composable with one another. Since tailorings run before the typechecker, host-language typechecking provides a basic correctness guarantee; for further assurance, this paper also presents a framework for reasoning about behavioral changes that tailorings may introduce. Finally, since a type tailoring leverages the host's metaprogramming system, no extra effort is needed to integrate it into a language's build system—unlike what a bespoke static analysis might require.

Although this paper has shown that support from a metaprogramming system empowers end users to build a type tailoring system, it would be interesting to see how first-class support for tailoring might improve its expressiveness and efficiency. No programming language includes type tailoring as an official, documented aspect of the language, but programmers clearly benefit from what support exists anyway. The many forms of tailoring in the programming language landscape give ample evidence that tailoring is a useful idea. The API blueprints presented here may allow these related efforts to build on one another.

In summary, this paper provides a research foundation that has takeaways for end users, language designers, and the authors of tailorings:

- For users, the examples in Section 2 show that type tailoring balances ease of use with some expressiveness of dependent types.
- For designers, the analysis in Section 3 explains why a powerful metaprogramming system is desirable. Tailoring in Julia, for example, falls short of what it could achieve with *cooperating* API features (Figure 1).
- For authors of tailorings, the guidelines in Section 5 separate well-reasoned tailorings from arbitrary reprogramming of the compiler front end.

Type tailoring leads to a broader perspective on what metaprogramming for types can achieve. It can facilitate maintainability and reliability for end users, help researchers prototype type system ideas, and reduce demands on the core type system.

References

1 Michael D. Adams. Towards the essence of hygiene. In *POPL*, pages 457–469. ACM, 2015. doi:10.1145/2676726.2677013.

2 Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards certified meta-programming with typed Template-Coq. In *ITP*, pages 20–39. Springer, 2018. doi:10.1007/978-3-319-94821-8_2.

3 Lennart Augustsson. Cayenne—a language with dependent types. In *ICFP*, pages 239–250. ACM, 1998. doi:10.1145/289423.289451.

4 Christiaan Baaij. GHC type checker plugins: adding new type-level operations, 2016. http://christiaanb.github.io/posts/type-checker-plugin/. Accessed 2016-06-30.

5 Langston Barrett, David Thrane Christiansen, and Samuel Gélineau. Predictable macros for Hindley–Milner. In *TyDE*, 2020. Extended abstract.

6 Gary Bernhardt. Software: static-path. https://github.com/garybernhardt/static-path. Accessed 2024-01-17.

7 Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 2017. doi:10.1137/141000671.

8 Gilad Bracha and David Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *OOPSLA*, pages 215–230, 1993. doi:10.1145/165854.165893.

9 Eugene Burmako. Scala macros: Let our powers combine!: On how rich syntax and static types work with metaprogramming. In *SCALA*, pages 3:1–3:10. ACM, 2013. doi:10.1145/2489837.2489840.

10 Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009. doi:10.1017/S0956796809007205.

11 Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. Dependent type systems as macros. *PACMPL*, 4(POPL):3:1–3:29, 2020. doi:10.1145/3371071.

12 Stephen Chang, Alex Knauth, and Ben Greenman. Type systems as macros. In *POPL*, pages 694–705. ACM, 2017. doi:10.1145/3009837.3009886.

13 David R. Christiansen and Edwin C. Brady. Elaborator reflection: Extending Idris in Idris. In *ICFP*, pages 284–297. ACM, 2016. doi:10.1145/2951913.2951932.

14 David Raymond Christiansen. Dependent type providers. In *WGP*, pages 25–34. ACM, 2013. doi:10.1145/2502488.2502495.

15 William D. Clinger and Jonathan Rees. Macros that work. In *POPL*, pages 155–162. ACM, 1991. doi:10.1145/99583.99607.

16 William D. Clinger and Mitchell Wand. Hygienic macro technology. *PACMPL*, 4(HOPL):80:1–80:110, 2020. doi:10.1145/3386330.

17 Clojure Contributors. Software: Clojure/tools.macro. https://github.com/clojure/tools.macro. Accessed 2024-01-18.

18 Clojure Contributors. Clojure macros, 2024. https://clojure.org/reference/macros. Accessed 2024-01-17.

19 Clojure Contributors. Clojure metadata documentation, 2024. https://clojure.org/reference/metadata. Accessed 2024-01-17.

20 Ryan Culpepper. Fortifying macros. *Journal of Functional Programming*, 22(4-5):439–476, 2012. doi:10.1017/S0956796812000275.

21 Ryan Culpepper, Sam Tobin-Hochstadt, and Matthew Flatt. Advanced macrology and the implementation of Typed Scheme. In *SFP. Université Laval, DIUL-RT-0701*, pages 1–14, 2007. URL: http://www2.ift.ulaval.ca/~dadub100/sfp2007/procPaper1.pdf.

22 Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kıvanç Muşlu, and Todd W. Schiller. Building and using pluggable type checkers. In *ICSE*, pages 681–690, 2011. doi:10.1145/1985793.1985889.

23 R. Kent Dybvig. *Chez Scheme Users Guide*. Cadence Research Systems, 2nd edition, 2011.

24 Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *PACMPL*, 1(ICFP):34:1–34:29, 2017. doi:10.1145/3110278.

25 ECMA International. ECMAScript language specification: 11.2.2 strict mode code, 2024. URL: https://262.ecma-international.org/15.0/index.html#sec-strict-mode-code.

26 Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *Haskell*, pages 117–130. ACM, 2012. doi:10.1145/2364506.2364522.

27 Elixir Contributors. Elixir macro documentation, 2024. https://hexdocs.pm/elixir/1.16/Macro.html. Accessed 2024-01-17.

28 Elixir Contributors. Elixir standard library, 2024. https://hexdocs.pm/elixir/1.16.0/Kernel.html. Accessed 2024-01-17.

29 Matthias Felleisen. Software: 7GUI, 2020. https://github.com/mfelleisen/7GUI. Accessed 2023-12-21.

30 David Fisher and Olin Shivers. Building language towers with Ziggurat. *Journal of Functional Programming*, 18(5-6):707–780, 2008. doi:10.1017/S0956796808006928.

31 Matthew Flatt. Composable and compilable macros: You want it when? In *ICFP*, pages 72–83. ACM, 2002. doi:10.1145/581478.581486.

32 Matthew Flatt. Binding as sets of scopes. In *POPL*, pages 705–717, 2016. doi:10.1145/2837614.2837620.

33 Matthew Flatt, Taylor Allred, Nia Angle, Stephen De Gabrielle, Robert Bruce Findler, Jack Firth, Kiran Gopinathan, Ben Greenman, Siddhartha Kasivajhula, Alex Knauth, Jay McCarthy, Sam Phillips, Sorawee Porncharoenwase, Jens Axel Søgaard, and Sam Tobin-Hochstadt. Rhombus: A new spin on macros without all the parentheses. *PACMPL*, 7(OOPSLA2), 2023. doi:10.1145/3622818.

34 Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. Macros that work together: Compile-time bindings, partial expansion, and definition contexts. *Journal of Functional Programming*, 22(2):181–216, 2012. doi:10.1017/S0956796812000093.

35 Kimball Germane, Jay McCarthy, Michael D. Adams, and Matthew Might. Demand control-flow analysis. In *VMCAI*, pages 226–246. Springer, 2019. doi:10.1007/978-3-030-11245-5_11.

36 GitHub. GitHub search: `use Phoenix.VerifiedRoutes` in Elixir, 2024. https://github.com/search?q=%22use+Phoenix.VerifiedRoutes%22+AND+%22def+verified_routes%22+language%3AElixir&type=code. Accessed 2024-01-17.

37 Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. In *ICFP*, pages 163–175. ACM, 2011. doi:10.1145/2034773.2034798.

38 Thomas R. G. Green and Marian Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996. doi:10.1006/JVLC.1996.0009.

39 Michael Greenberg. The dynamic practice and static theory of gradual typing. In *SNAPL*, pages 6:1–6:20. Schloss Dagstuhl, 2019. doi:10.4230/LIPICS.SNAPL.2019.6.

40 Ben Greenman. Trivial: Type tailored library functions, 2020. https://docs.racket-lang.org/trivial/index.html. Accessed 2024-01-07.

41 Ben Greenman. GTP benchmarks for gradual typing performance. In *REP*, pages 102–114. ACM, 2023. doi:10.1145/3589806.3600034.

42 Adam Gundry. A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. In *Haskell*, pages 11–22. ACM, 2015. doi:10.1145/2804302.2804305.

43 Susumu Hayashi. Singleton, union and intersection types for program extraction. *Information and Computation*, 109(1/2):174–210, 1994. doi:10.1006/INCO.1994.1016.

44 David Herman and Philippe Meunier. Improving the static analysis of embedded languages via partial evaluation. In *ICFP*, pages 16–27. ACM, 2004. doi:10.1145/1016850.1016857.

45 Joel Jakubovic, Jonathan Edwards, and Tomas Petricek. Technical dimensions of programming systems. *Programming*, 7(3), 2023. doi:10.22152/PROGRAMMING-JOURNAL.ORG/2023/7/13.

46 Guy L. Steele Jr. Growing a language. In *Addendum to OOPSLA*. ACM, 1998. doi:10.1145/346852.346922.

47 Julia Contributors. Julia AST documentation, 2024. https://docs.julialang.org/en/v1/devdocs/ast/. Accessed 2024-01-17.

48 Julia Contributors. Julia metaprogramming, 2024. https://docs.julialang.org/en/v1/manual/metaprogramming/. Accessed 2024-01-17.

49 Milod Kazerounian, Sankha Narayan Guria, Niki Vazou, Jeffrey S. Foster, and David Van Horn. Type-level computations for Ruby libraries. In *PLDI*, pages 966–979. ACM, 2019. doi:10.1145/3314221.3314630.

50 Daniel Keep and Lukas Wirth. The little book of Rust macros, 2024. https://veykril.github.io/tlborm/proc-macros/hygiene.html. Accessed 2024-01-17.

51 Oleg Kiselyov. Reconciling abstraction with high performance: A MetaOCaml approach. *Foundations and Trends® in Programming Languages*, 5(1):1–101, 2018. doi:10.1561/2500000038.

52 Eugen Kiss. 7GUIs: A GUI programming benchmark. https://eugenkiss.github.io/7guis/. Accessed 2023-12-21.

53 Eugen Kiss. *Comparison of Object-Oriented and Functional Programming for GUI Development*. PhD thesis, Leibniz Universitaet Hannover, 2014.

54 Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce F. Duba. Hygienic macro expansion. In *LFP*, pages 151–161. ACM, 1986. doi:10.1145/319838.319859.

55 Eugene E. Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *POPL*, pages 77–84. ACM, 1987. doi:10.1145/41625.41632.

56 Pepijn Kokke and Wouter Swierstra. Auto in Agda - programming proof search using reflection. In *MPC*, pages 276–301. Springer, 2015. doi:10.1007/978-3-319-19797-5_14.

57 Joomy Korkut and David Thrane Christiansen. Extensible type-directed editing. In *TyDe*, pages 38–50. ACM, 2018. doi:10.1145/3240719.3241791.

58 LaunchBadge. Software: SQLx, 2023. https://github.com/launchbadge/sqlx. Accessed 2024-01-17.

59 Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in Agda. In *TYPES*, pages 154–169. Springer, 2004. doi:10.1007/11617990_10.

60 Yiyun Liu and Stephanie Weirich. Dependently-typed programming with logical equality reflection. *PACMPL*, 7(ICFP):210:1–210:37, 2023. doi:10.1145/3607852.

61 Florian Lorenzen and Sebastian Erdweg. Sound type-dependent syntactic language extension. In *POPL*, pages 204–216. ACM, 2016. doi:10.1145/2837614.2837644.

62 Gregory Malecha and Jesper Bengtson. Extensible and efficient automation through reflective tactics. In *ESOP*, pages 532–559. Springer, 2016. doi:10.1007/978-3-662-49498-1_21.

63 Michał Marczyk. Answer to "does Clojure have identifier macros?". https://stackoverflow.com/a/33426863/7327755. Accessed 2024-01-18.

64 Chris McCord. Phoenix 1.7.0 released: Built-in Tailwind, Verified Routes, LiveView Streams, and what's next, 2023. https://phoenixframework.org/blog/phoenix-1.7-final-released. Accessed 2024-01-17.

65 Alp Mestanogullari, Sönke Hahn, Julian K. Arni, and Andres Löh. Type-level web APIs with Servant: An exercise in domain-specific generic programming. In *WGP*, pages 1–12. ACM, 2015. doi:10.1145/2808098.2808099.

66 Ana L. Milanova and Wei Huang. Inference and checking of context-sensitive pluggable types. In *FSE*, page 26. ACM, 2012. doi:10.1145/2393596.2393626.

67 David A. Moon. MACLISP reference manual, Revision 0. Technical report, MIT Project MAC, 1974.

68 Multiple Authors. Language: Macros, 2023. https://clojure-doc.org/articles/language/macros/. Accessed 2024-01-17.

69 OCaml Contributors. OCaml PPX, 2024. https://ocaml.org/docs/metaprogramming. Accessed 2024-01-17.

70 Cyrus Omar, Darya Kurilova, Ligia Nistor, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. Safely composable type-specific languages. In *ECOOP*, pages 105–130. Springer, 2014. doi:10.1007/978-3-662-44202-9_5.

71 Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA*, pages 201–212. ACM, 2008. doi:10.1145/1390630.1390656.

72 Lionel Parreaux. Squid—type-safe metaprogramming for Scala, 2024. https://epfldata.github.io/squid/home.html. Accessed 2024-01-17.

73 Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. Unifying analytic and statically-typed quasiquotes. *PACMPL*, 2(POPL):13:1–13:33, 2018. doi:10.1145/3158101.

74 Tomas Petricek, Gustavo Guerra, and Don Syme. Types from data: Making structured data first-class citizens in F#. In *PLDI*, pages 477–490. ACM, 2016. doi:10.1145/2908080.2908115.

75 Phoenix Contributors. Phoenix verified routes, 2023. https://hexdocs.pm/phoenix/Phoenix.VerifiedRoutes.html. Accessed 2023-11-28.

76 Justin Pombrio and Shriram Krishnamurthi. Hygienic resugaring of compositional desugaring. In *ICFP*, pages 75–87. ACM, 2015. doi:10.1145/2784731.2784755.

77 Racket Contributors. Racket syntax properties documentation, 2024. https://docs.racket-lang.org/reference/stxprops.html. Accessed 2024-01-17.

78 Racket Contributors. Syntax transformers, 2024. https://docs.racket-lang.org/reference/stxtrans.html. Accessed 2024-01-17.

79 Rhombus Contributors. Rhombus documentation: Static and dynamic lookup, 2024. https://docs.racket-lang.org/rhombus/Static_and_Dynamic_Lookup.html. Accessed 2024-01-17.

80 Rhombus Contributors. Rhombus syntax object documentation, 2024. https://docs.racket-lang.org/rhombus/stxobj.html. Accessed 2024-01-17.

81 Tiark Rompf. Reflections on LMS: exploring front-end alternatives. In *SCALA*, pages 41–50. ACM, 2016. doi:10.1145/2998392.2998399.

82 Tiark Rompf and Nada Amin. A SQL to C compiler in 500 lines of code. *Journal of Functional Programming*, 29:e9, 2019. doi:10.1017/S0956796819000054.

83 Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*, pages 127–136. ACM, 2010. doi:10.1145/1868294.1868314.

84 Tiark Rompf, Arvind K. Sujeeth, Nada Amin, Kevin J. Brown, Vojin Jovanovic, HyoukJoong Lee, Manohar Jonnalagedda, Kunle Olukotun, and Martin Odersky. Optimizing data structures in high-level programs: New directions for extensible compilers based on staging. In *POPL*, pages 497–510. ACM, 2013. doi:10.1145/2429069.2429128.

85 Rust Contributors. Rust macros, 2024. https://doc.rust-lang.org/reference/procedural-macros.html. Accessed 2024-01-17.

86 Scala 3 Contributors. Scala 3 macros, 2024. https://docs.scala-lang.org/scala3/guides/macros/macros.html. Accessed 2024-01-17.

87 Scala 3 Contributors. Scala 3 reference: Macros, 2024. https://docs.scala-lang.org/scala3/reference/metaprogramming/macros.html. Accessed 2024-01-17.

88 SciML Contributors. Software: OrdinaryDiffEq.jl, 2024. https://github.com/SciML/OrdinaryDiffEq.jl. Accessed 2024-01-17.

89 Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Haskell*, pages 1–16. ACM, 2002. doi:10.1145/581690.581691.

90 Olin Shivers. Control-flow analysis in Scheme. In *PLDI*, pages 164–174. ACM, 1988. doi:10.1145/53990.54007.

91 Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *SFP. University of Chicago, TR-2006-06*, pages 81–92, 2006. URL: http://scheme2006.cs.uchicago.edu/scheme2006.pdf.

92 Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. Refined criteria for gradual typing. In *SNAPL*, pages 274–293. Schloss Dagstuhl, 2015. doi:10.4230/LIPICS.SNAPL.2015.274.

93 Suzanne Soy. Software: `type-expander`, 2020. `https://github.com/SuzanneSoy/type-expander`. Accessed 2023-12-21.

94 Eric Spishak, Werner Dietl, and Michael D. Ernst. A type system for regular expressions. In *FTfJP*, pages 20–26. ACM, 2012. `doi:10.1145/2318202.2318207`.

95 Antonis Stampoulis and Zhong Shao. VeriML: Typed computation of logical terms inside a language with effects. In *ICFP*, pages 333–344. ACM, 2010. `doi:10.1145/1863543.1863591`.

96 StaticArrays Contributors. StaticArrays, 2024. `https://juliahub.com/ui/Packages/General/StaticArrays/`. Accessed 2024-01-17.

97 Guy L. Steele, Jr. *Common Lisp*. Digital Press, 2nd edition, 1990.

98 T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical variable-arity polymorphism. In *ESOP*, pages 32–46, 2009. `doi:10.1007/978-3-642-00590-9_3`.

99 Satish Thatte. Quasi-static typing. In *POPL*, pages 367–381, 1990. `doi:10.1145/96709.96747`.

100 Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *DLS*, pages 964–974, 2006. `doi:10.1145/1176617.1176755`.

101 Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL*, pages 395–406, 2008. `doi:10.1145/1328438.1328486`.

102 Sam Tobin-Hochstadt, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Ben Greenman, Andrew M. Kent, Vincent St-Amour, T. Stephen Strickland, and Asumu Takikawa. Migratory typing: Ten years later. In *SNAPL*, pages 17:1–17:17. Schloss Dagstuhl, 2017. `doi:10.4230/LIPICS.SNAPL.2017.17`.

103 Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *PLDI*, pages 132–141, 2011. `doi:10.1145/1993498.1993514`.

104 José Valim. Nx: Numerical Elixir, 2023. `https://github.com/elixir-nx/nx`. Accessed 2024-01-16.

105 David Van Horn. Software: `zombie`, 2020. `https://github.com/philnguyen/soft-contract/tree/master/soft-contract/benchmark-contract-overhead`. Accessed 2023-02-20.

106 Stephanie Weirich. The influence of dependent types (keynote). *SIGPLAN Notices*, 52(1), 2017. `doi:10.1145/3093333.3009923`.

107 Ashton Wiersdorf, Stephen Chang, Matthias Felleisen, and Ben Greenman. Artifact for Type Tailoring (ECOOP 2024), July 2024. `doi:10.5281/zenodo.12726060`.

108 Hongwei Xi. Dependent ML: An approach to practical programming with dependent types. *Journal of Functional Programming*, 17(2):215–286, 2007. `doi:10.1017/S0956796806006216`.

109 Ningning Xie, Leo White, Olivier Nicole, and Jeremy Yallop. MacoCaml: Staging composable and compilable macros. *PACMPL*, 7(ICFP), 2023. `doi:10.1145/3607851`.