Type Tailoring

Ben Greenman, Stephen Chang, and Matthias Felleisen

Northeastern University
Boston, Mass.
{types,stchang,matthias}@ccs.neu.edu

Abstract. *Type tailoring* enables programmers to extend the rules of a type *system without modifying the compiler*. Programmers may thus codify additional knowledge about functions and methods in the form of a library of typing rules, which others may use without imposing any additional burden on their code or work flow. Type tailoring relies on an API to the type elaborator, the compiler pass that maps the surface syntax to a core calculus. This paper presents a blueprint for such an API, describes a prototype written in Typed Racket, and demonstrates the usefulness of type tailoring with two case studies.

1 Tailoring Types

Type systems often force programmers to assign overly conservative types to functions and methods. For instance, most programming languages support a function for matching regular expressions with strings, and the library might assign the following Haskellian type to such a function:

$$\texttt{rx-match} : \mathsf{String}\ \mathsf{String} \to \mathsf{Maybe}\,[\mathsf{String}]$$

The first argument string represents the pattern and the second the input; the purpose of the function is to find instances of the former in the latter. But clearly, the pattern cannot be any arbitrary string. More importantly still, a successful match produces a list of strings whose length *depends* on the pattern.

A dependent-type system could express this relationship with a reasonably precise type. Until we reach paradise where all programmers use languages with dependent-type systems for all software development projects, programmers and/or compilers must circumvent the existing "simple" type systems with extraneous runtime checks to cope with just such situations.

This paper presents an alternative that many modern languages can implement without imposing a new type discipline on programmers at all, *type tailoring*. The basic idea is to empower programmers so that they can refine the rules of a type system without modifying the compiler. For example, the creator of `rx-match` may add this rule:[1]

RX-REFINE

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \mathsf{String} \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \mathsf{String} \qquad \textit{Program} \vdash e_1' \text{ is a valid pattern and contains } n \text{ groupings}}{\Gamma \vdash \texttt{rx-match}\ e_1\ e_2 \rightsquigarrow \texttt{rx-match}\ e_1'\ e_2' : \mathsf{Maybe}\,[\mathsf{String}]_{n+1}}$$

[1] User-defined *tailorings* are highlighted in green throughout the paper.

The rule extends the type elaborator so that if it can prove that a program's specific use of `rx-match` has a valid pattern containing *n* grouping specifications—ways to extract a sub-string via matching—the Some result type is a *fixed-length* list containing the matched string plus exactly *n* substrings corresponding to the group specifications. Since this new type information ripples through the rest of the type elaboration process, the compiler can replace access operations such as `List.head` with *unsafe* (and thus faster) versions. Best of all, this improvement comes at no cost to the end user; if the program does not contain enough information to use the refined rule, the type checker silently falls back on the default rule.

A type tailoring requires an API for the type elaborator pass in the compiler that satisfies specific requirements (section 5), all motivated with concrete examples (sections 2 through 4). The paper also sketches how to implement this "blueprint" in Typed Racket (section 6). Case studies based on this first prototype (section 7) suggest that type tailoring has tremendous potential and deserves further exploration.

2 A Glimpse of an Elaborator API

A typical elaborator maps the surface syntax of a programming language to the syntax of some core "calculus." In the process, it simultaneously synthesizes a fully annotated, abstract syntax tree and checks this tree against the rules of the type system. One variant is to check types before the syntax is elaborated into kernel syntax [15]; another one is to elaborate only and to type check the resulting syntax tree [3, 13, 18].

Type tailoring relies on the latter approach. The idea is to give programmers an API to create new type elaboration rules and type check only the result of elaboration.[2] Such rules can mediate between the syntax exposed to programmers and the core language of the type checker, for instance, by inserting type annotations, interpreting domain-specific syntax, or rewriting code.

Figure 1 presents a toy example: a simply-typed λ calculus with integer arrays. The syntax of this toy language is divided into a surface syntax (for programming) and a core syntax (for type checking). Surface and core are nearly identical; however, the surface language provides syntactic sugar for multi-argument functions and a variable-arity (array) `map` function that does not have a simple type. In contrast, the core language has primitives $\texttt{map}_1$ and $\texttt{map}_2$ for mapping over one or two arrays.

The typed elaboration relation $\Gamma \vdash e \rightsquigarrow e' \cup \mathsf{Map\text{-}Exn} : \tau, v^{\mathcal{A}}$ bridges the gap between the (library writer's) surface syntax and (compiler writer's) core syntax. The relation states that the surface term e elaborates to the core term e' (or Map-Exn) and furthermore asserts that e' has type τ and arity $v^{\mathcal{A}}$.[3]

The four tailorings in figure 2 mediate between the surface and core languages. The rule T-CURRY elaborates a multi-argument function to curried form and also synthesizes the function's arity information. The rules for `map` check this stored arity and the number of additional arguments passed to `map`. If the arity and number of arrays are both 1 or both 2, then `map` elaborates to a core form.

[2] This approach is reminiscent of *translation validation* [17].

[3] The domain $v^{\mathcal{A}}$ is the flat lattice of natural numbers. When $v^{\mathcal{A}}$ is $\bot$, the arity is unknown. When it is $\top$, expression is not a function.

Common Syntax

$$
\begin{aligned}
e &= v \mid x \mid e\ e \mid \langle e_0, \ldots, e_{n-1} \rangle \\
v &= \mathbb{Z} \mid \langle v_0, \ldots, v_{n-1} \rangle \\
\tau &= \mathsf{Int} \mid \mathsf{Array} \mid \tau \to \tau \\
\Gamma &= \cdot \mid x{:}\tau, \Gamma \\
v^{\mathcal{A}} &= \bot \mid \mathbb{N} \mid \top
\end{aligned}
$$

Surface Syntax

(extends common syntax)

$$
\begin{aligned}
e &= \ldots \mid \lambda^{+}\, x_0{:}\tau_0 \ldots x_{n-1}{:}\tau_{n-1}\,.\,e \\
&\quad \mid \mathtt{map}\ e_0\ e_1\ \ldots\ e_n \\
v &= \lambda\, x{:}\tau\,.\,e
\end{aligned}
$$

Core Syntax

(extends common syntax)

$$
\begin{aligned}
e &= \ldots \mid \mathtt{map}_1\ e\ e \\
&\quad \mid \mathtt{map}_2\ e\ e\ e \\
v &= \lambda\, x\,.\,e
\end{aligned}
$$

Figure 1: A simply typed language with integer arrays

$\boxed{\Gamma \vdash e \rightsquigarrow e \cup \mathsf{Map\text{-}Exn} : \tau,\ v^{\mathcal{A}}}$

T-CURRY

$$\frac{x_0{:}\tau_0, \ldots, x_{n-1}{:}\tau_{n-1}, \Gamma \vdash e \rightsquigarrow e' : \tau',\ v^{\mathcal{A}}}{\Gamma \vdash \lambda^{+}\, x_0{:}\tau_0 \ldots x_{n-1}{:}\tau_{n-1}\,.\,e \rightsquigarrow \lambda\, x_0\,.\,\ldots\,\lambda\, x_{n-1}\,.\,e' : \tau_0 \to \ldots \to \tau_{n-1} \to \tau',\ n}$$

T-MAP1

$$\frac{\Gamma \vdash e_0 \rightsquigarrow e_0' : \mathsf{Int} \to \mathsf{Int},\ 1 \qquad \Gamma \vdash e_1 \rightsquigarrow e_1' : \mathsf{Array},\ \top}{\Gamma \vdash \mathtt{map}\ e_0\ e_1 \rightsquigarrow \mathtt{map}_1\ e_0'\ e_1' : \mathsf{Array},\ \top}$$

T-MAP2

$$\frac{\Gamma \vdash e_0 \rightsquigarrow e_0' : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Int},\ 2 \qquad \Gamma \vdash e_1 \rightsquigarrow e_1' : \mathsf{Array},\ \top \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \mathsf{Array},\ \top}{\Gamma \vdash \mathtt{map}\ e_0\ e_1\ e_2 \rightsquigarrow \mathtt{map}_2\ e_0'\ e_1'\ e_2' : \mathsf{Array},\ \top}$$

T-MAPERR

$$\frac{\Gamma \vdash e_0 \rightsquigarrow e_0' : \tau,\ v^{\mathcal{A}} \qquad v^{\mathcal{A}} \notin \{1,2\} \quad \text{or} \quad n \notin \{1,2\} \quad \text{or} \quad v \neq n}{\Gamma \vdash \mathtt{map}\ e_0\ \cdots\ e_n \rightsquigarrow \mathsf{Map\text{-}Exn} : \mathsf{Array},\ \top}$$

Figure 2: Type-tailored, variable-arity `map`

Note: the arity of a function is evident from its type. Figure 2 tracks arity separately because (1) type information may not be available *during* elaboration and (2) to introduce a notation for tracking arbitrary propositions.

With an API to the type elaborator, library authors can themselves implement many such rules that use a lightweight static analysis to justify program transformations. In other words, such an API empowers programmers to *grow* the type system [12]. For instance, a straightforward extension of figure 2 could track the length of arrays and raise a static error if `map` receives two array arguments of unequal length. The same extension could encode the knowledge that `map` is a length-preserving operation.

In general, type tailoring has three goals:

- ***Refine*** type signatures using facts manifest in the syntax of a program.
- ***Reuse*** the existing type checker and accomodate legacy code.[4]
- Point to ***relevant*** surface syntax in error messages, rather than elaborated code.[5]

The author of new tailorings (elaboration rules) must prove that the overall system remains terminating and sound. While the correctness of `map` is easy to argue, an API to the type elaborator calls for general strategies to prove particular uses correct. Experience with a Typed Racket prototype has identified two kinds of cases. One kind of type tailoring may introduce unsafe behavior (segfaults); section 3 deals with this case. Others give library authors a technique for merely lowering API types; section 4 is about this second kind of type tailorings.

3 When Using the API May Cause a SEGFAULT

Every programming languages comes with arrays. In Typed Racket, array facilities come as a library that exports constructors, dereferencing functions, and mutation operations. SML provides similar facilities from a run-time library [10]. The APIs for array libraries tend to come with conservative signatures. For example, the type for an array indexing operation calls for an array and an integer; run-time checks ensure that the integer belongs to the domain of the array before any memory access occurs.

With an elaborator API, the array library can be improved in two ways. First, it might be able to prove statically that an index is out-of-bounds for the given array. While rare, this catch turns a run-time error into a type error. Second, if it can prove that an index is within the bounds for the array, it is possible to remove the run-time bounds check and thus speed up the program execution.

This second scenario may ring alarm bells with an alert reader. When programmers can program the type system, a sound system can turn into an unsound, C++-like type system where programs compute nonsense or crash with a segfault. At a minimum, an elaborator API must come with a method for extending a standard type soundness proof so that programmers can prove the absence of segfaults in such a tailored type system. Section 3.1 presents one such proof method with a simple example.

[4] Type tailoring is *not* about building typed, embedded DSLs. There are much better techniques for building embedded languages from scratch [2].

[5] The SoundX system [15] pursues the same goal with different means and re-enforced the Racket lessons [7] for this project.

An evaluation of the Racket code base shows this primitive evaluator can effectively check and optimize useful code (section 6.3). But the point of this section is not to reinvent an optimization that every modern compiler implements. Instead, its purpose is to illustrate with an example that everyone knows, what the complete use of the elaborator API looks like. This section uses mathematical notation; actual code can be found in the appendix.

3.1 Array Indexing

Any attempt to model array indexing in a potentially unsafe context must start from a relation that, like a C++-style language, represents invalid array access as the act of retrieving (and misinterpreting) arbitrary bits from memory.

The model in figure 3 extends the common syntax of figure 1 with an array indexing operation. The middle portion of the figure specifies the type-directed elaboration of surface terms into a typed, executable core language (omitting the syntax-directed rules). The translation replaces array indexing expressions with applications of `checked-ref`, a primitive operation that dynamically checks the safety of the array access. For example, the term $x@3$ elaborates to `checked-ref` x 3 when x is a variable with type Array. The core syntax also comes with an `unsafe-ref` primitive but the standard elaboration does not synthesize expressions that use this dangerous function.

Figure 3 also specifies the operational semantics of the core language. Not surprisingly, `checked-ref` raises a runtime error when called with incompatible values. In contrast, `unsafe-ref` does not perform an array-bounds check. If the index is out of bounds, it may produce a whimsically chosen integer or segfault, reflecting the non-deterministic behavior of an unsafe memory access.[6] This behavior, modeled by E-UnsafeBits and E-UnsafeSeg, makes $e \longrightarrow e$ a proper relation. Nevertheless, the elaboration in figure 3 creates a deterministic program.

> **Definition (*evaluation*):** *eval* relates core e to answers a. e *eval* a iff e is closed and there exists a term e' such that $\cdot \vdash e \rightsquigarrow e' : \mathsf{Int}$, $v^{\mathcal{V}}$ and $e' \longrightarrow^* a$.

> **Theorem (*determinism*):** *eval* is a function.

Proof: The surface language does not include `unsafe-ref` and the elaboration rules do not introduce unsafe references. □

With determinism in hand, type soundness follows.

> **Theorem (*soundness*):** For all programs e, one of the following holds:
> – there exists an integer v such that e *eval* v
> – e *eval* Index-Exn
> – *eval* is undefined for e (e goes into an infinite loop).

Proof idea: The elaboration rules define a bijective function. Hence it is possible to work out a conventional progress-preservation proof, with plain type checking proof trees replaced by type elaboration proof trees. (Details in Appendix) □

[6] This is somewhat inaccurate. In an actual language, a segfault is triggered by a flow of misinterpreted bits into a computational operation that cannot cope, not necessarily just the array access itself. The model is too sparse to express all possible problems of C++.

Surface Syntax
(extends common syntax)

$$e = \ldots \mid e@e \mid \lambda\, x{:}\tau\,.\, e$$

Core Syntax
(extends common syntax)

$$e = \ldots \mid \texttt{checked-ref}\ e\ e \mid \texttt{unsafe-ref}\ e\ e$$

$$v = \ldots \mid \lambda\, x\,.\, e$$

Runtime Syntax
(extends core syntax)

$$a = v \mid \textsf{Index-Exn} \mid \textsf{segfault}$$

$$E = \ldots \mid \langle v, \ldots, E, e, \ldots \rangle \mid \texttt{checked-ref}\ E\ e \mid \texttt{checked-ref}\ v\ E \mid \texttt{unsafe-ref}\ E\ e \mid \texttt{unsafe-ref}\ v\ E$$

$$v^{\mathcal{V}} = \bot \mid \mathbb{N} \mid \top$$

$\boxed{\Gamma \vdash e \rightsquigarrow e \cup a : \tau,\ v^{\mathcal{V}}}$

T-Var
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \rightsquigarrow x : \tau,\ \bot}$$

T-Int
$$\frac{n \in \mathbb{Z}}{\Gamma \vdash n \rightsquigarrow n : \textsf{Int},\ \top}$$

T-App
$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \tau \to \tau',\ v_1^{\mathcal{V}} \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau,\ v_2^{\mathcal{V}}}{\Gamma \vdash e_1\ e_2 \rightsquigarrow e_1'\ e_2' : \tau',\ \bot}$$

T-Arr
$$\frac{\Gamma \vdash e_0 \rightsquigarrow e_0' : \textsf{Int},\ \top \quad \ldots \quad \Gamma \vdash e_{n-1} \rightsquigarrow e_{n-1}' : \textsf{Int},\ \top}{\Gamma \vdash \langle e_0, \ldots, e_{n-1} \rangle \rightsquigarrow \langle e_0', \ldots, e_{n-1}' \rangle : \textsf{Array},\ n}$$

T-Lam
$$\frac{x{:}\tau, \Gamma \vdash e \rightsquigarrow e' : \tau',\ v^{\mathcal{V}}}{\Gamma \vdash \lambda\, x{:}\tau\,.\, e \rightsquigarrow \lambda\, x\,.\, e' : \tau \to \tau',\ \bot}$$

T-Ref=
$$\frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \textsf{Array},\ v^{\mathcal{V}} \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \textsf{Int},\ \bot}{\Gamma \vdash e_1@e_2 \rightsquigarrow \texttt{checked-ref}\ e_1'\ e_2' : \textsf{Int},\ \bot}$$

$\boxed{e \to e \cup a}$

E-App
$$E[(\lambda\, x\,.\, e)\ v] \to E[e[v/x]]$$

E-Check
$$\frac{0 \leqslant i < n}{E[\texttt{checked-ref}\ \langle v_0, \ldots, v_{n-1} \rangle\ i] \to E[v_i]}$$

E-CheckErr
$$\frac{i < 0 \ \text{ or } \ i \geqslant n}{E[\texttt{checked-ref}\ \langle v_0, \ldots, v_{n-1} \rangle\ i] \to \textsf{Index-Exn}}$$

E-Unsafe
$$\frac{0 \leqslant i < n}{E[\texttt{unsafe-ref}\ \langle v_0, \ldots, v_{n-1} \rangle\ i] \to E[v_i]}$$

E-UnsafeBits
$$\frac{i < 0 \ \text{ or } \ i \geqslant n \qquad v \in \mathbb{Z}}{E[\texttt{unsafe-ref}\ \langle v_0, \ldots, v_{n-1} \rangle\ i] \to E[v]}$$

E-UnsafeSeg
$$\frac{i < 0 \ \text{ or } \ i \geqslant n}{E[\texttt{unsafe-ref}\ \langle v_0, \ldots, v_{n-1} \rangle\ i] \to \textsf{segfault}}$$

Figure 3: Elaboration and sematics of array indexing

$$\textsc{T-Ref}^{+}\ \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \mathsf{Array},\, n \qquad \Gamma \vdash e_2 \rightsquigarrow i : \mathsf{Int},\, \bot \qquad i \in \mathbb{Z} \qquad 0 \leqslant i < n}{\Gamma \vdash e_1 @ e_2 \rightsquigarrow \texttt{unsafe-ref}\ e_1'\ i : \mathsf{Int},\, \bot}$$

$$\textsc{T-Ref}^{-}\ \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \mathsf{Array},\, n \qquad \Gamma \vdash e_2 \rightsquigarrow i : \mathsf{Int},\, \bot \qquad i \in \mathbb{Z} \qquad i < 0 \text{ or } i \geqslant n}{\Gamma \vdash e_1 @ e_2 \rightsquigarrow \mathsf{Index\text{-}Exn} : \mathsf{Int},\, \bot}$$

Figure 4: Type-tailored array indexing

3.2 Type-Tailored Array Indexing

Figure 4 shows tailorings that can (1) generate an application of `unsafe-ref` when the index expression is provably within bounds and (2) an Index-Exn when the index is provably out of bounds.[7]

> **Definition (*evaluation, revised*):** $e\ \mathit{eval}^u\ a$ iff e is closed and there exists e' such that $\cdot \vdash e \rightsquigarrow^u e' : \mathsf{Int},\, v^{\mathcal{V}}$ (meaning e' is not Index-Exn) and $e' \rightarrow^* a$. The relation $\rightsquigarrow^u$ is the union of the rules in figures 3 and 4.

Adding these rules does not undermine the safety guarantees of the original language. In particular, evaluation cannot end in a random integer or cause a segfault, and the type system correctly predicts the behavior of the programs.

> **Theorem (*determinism & soundness, revised*):** eval^u is a function. For all programs e, one of the following holds:
> – there exists an integer v such that $e\ \mathit{eval}^u\ v$
> – $e\ \mathit{eval}^u$ Index-Exn
> – eval^u is undefined for e.

Proof idea: The extended type elaboration inference system may produce one of two different core terms for a given array indexing term from the surface language. If the derivation tree uses T-Ref$^+$, progress and preservation follow because the rule produces `unsafe-ref` $\langle v_0, \ldots, v_{n-1} \rangle\, i$, only when the premise $0 \leqslant i < n$ is met, which rules out the use of the non-deterministic rules E-UnsafeBits and E-UnsafeSeg.

Given this insight, it is possible the take the existing proofs of the meta-theorems and to extend their case analysis with one additional case. The substitution lemma also requires additional cases for the new core syntax. (Details in Appendix) □

The key take-away is that the proofs of the revised meta-theorems can be obtained by systematic surgery on the proofs of the original meta-theorems. Hence if a type soundness proof exists for the core language, a programmer can argue the type soundness of additional type tailorings. Such proofs would be impossible with traditional, evidence-free type judgments of the form $\Gamma \vdash$ `unsafe-ref` $e\ e : \tau$. It is elaboration that supplies the necessary evidence and interposition to justify unsafe operations.

[7] Values in $v^{\mathcal{V}}$ represent array (vector) bounds.

4 When Using the API Merely Lowers the Type

Many, if not most, programming languages come with libraries for regular-expression matching. Historically, these libraries represent regular expressions as strings, and modern languages continue to do so.

The designers of such libraries must accept the type system of the host language as-is, therefore dependencies between the input pattern string and result of regular expression matching are expressed only informally in most APIs. With type tailoring, designers are able to lift these dependencies into the type system.

As indicated in the introduction, the client code of a regular-expression library can benefit from a type tailoring that uses the number of parenthesized groups in a regular expression pattern to lower the return type of the matching operation. Doing so requires a notion of subtyping, so that lowered results are compatible with existing code. In Typed Racket, fixed-length lists are a subtype of arbitrarily long lists; the prototype described in section 6.3 uses this facility. Other typed languages can use records or objects to represent matches. The model in section 4.1 takes this approach.

The implementation of type tailoring for `regexp-match` in Typed Racket validates that an elaborator API is well worth the effort (see section 6.3).

4.1 Regular Expressions

Figure 5 is a simply-typed languages with strings and records (with *width* subtyping). It also mixes in a regular-expression matching primitive, dubbed `rx-match`. The elaboration rules track the number of matched parentheses in string values using the domain $v^{\mathcal{R}}$, though none of the rules in figure 5 access this information.[8]

Patterns p are strings that `rx-match` interprets as automata. The language $\mathcal{L}(p)$ recognized by a pattern string is either p itself, if p does not contain parentheses ($($ $)$) or stars ($\star$), or the set of strings with zero or more repetitions of characters in p preceded by stars. Parentheses in p are filtered from $\mathcal{L}(p)$.

Convention: The β sub-patterns come with labels so that successful matches can create records with recognizable field names. Contexts where β plays the role of a string ignore these labels. □

Figure 6 articulates the formal semantics of the regular-expression matching primitive. All other expressions have the usual semantics; see figure 3.

When called with a pattern p and a string s, the `rx-match` function checks whether any sub-string of s is in $\mathcal{L}(p)$. If so, `rx-match` chooses a canonical matching sub-string (for instance, the longest matching sub-string) using sup. After selecting a canonical match s', parentheses in p are used to extract sub-strings of s'. To keep the semantics of matching and grouping-by-parentheses simple, `rx-match` fails when parentheses in p are unbalanced, nested, or followed by a star. Due to this restriction, it is straightforward to express p as the concatenation of non-capturing (α) and capturing (β) patterns. This concatenation is matched pointwise against all tuples $\mathcal{C}^{2n+1}(s')$ obtained by splitting s' into a concatenation of $2n + 1$ (possibly empty) sub-strings. Finally sup selects a canonical matching tuple.

[8] In particular, an element of $v^{\mathcal{R}}$ is drawn from the lattice of label sequences.

Regexp Syntax

$$
\begin{array}{lll}
e & = v \mid e\ e \mid \texttt{rx-match}\ e\ e & \\
v & = \lambda x\,.\,e \mid \mathbb{C}^* & \\
 & \quad \mid \{l = v, \ldots, l = v\} & \\
 & \quad \mid \mathsf{None} \mid \mathsf{Some}\, v & \\
\tau & = \tau \to \tau \mid \mathsf{String} & \\
 & \quad \mid (\!|\, l\colon \tau, \ldots, l\colon \tau \,|\!) & \\
 & \quad \mid \mathsf{Maybe}\, \tau & \\
v^{\mathcal{R}} & = \bot \mid \langle l_0, \ldots, l_{n-1} \rangle \mid \top & \\
\mathbb{C} & : \text{ ANSI characters (: reserved)} & \\
l & : \text{ labels } (l^{\#} \text{ reserved}) &
\end{array}
$$

$$
\begin{array}{lll}
p & = \{\, s \mid s \in \mathbb{C}^* \wedge s = \alpha\beta\alpha\cdots\beta\alpha \,\} & \\
\alpha & = \{\, \varepsilon \,\} \cup \{\, cs \mid c \in \mathbb{C} \setminus \{\, (\,) \star \,\} & \\
 & \qquad \wedge\, s \in (\mathbb{C} \setminus \{\, (\,) \,\})^* \,\} & \\
\beta & = \{\, \varepsilon \,\} \cup \{\, (l\colon s) \mid s \in \alpha \,\} &
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{L}(\varepsilon) & = \{\, \varepsilon \,\} \\
\mathcal{L}((s)) & = \mathcal{L}(s) \\
\mathcal{L}(c \star s) & = \{\, c^k s' \mid k \geqslant 0 \wedge s' \in \mathcal{L}(s) \,\} \\
\mathcal{L}(cs) & = \{\, cs' \mid s' \in \mathcal{L}(s) \,\}
\end{array}
$$

$$
\begin{array}{ll}
\mathcal{S}(s) & = \{\, s_1 \mid s = s_0 s_1 s_2 \,\} \\
\mathcal{C}^k(s) & = \{\, (s_0, \ldots, s_{k-1}) \mid s = s_0 \cdots s_{k-1} \,\}
\end{array}
$$

$\boxed{\tau \leqslant: \tau}$

$$\text{S-Refl}\quad \tau \leqslant: \tau \qquad\qquad \text{S-Top}\quad (\!|\, l_0\colon \tau_0, \ldots, l_{n-1}\colon \tau_{n-1} \,|\!) \leqslant: (\!|\ |\!)$$

$$\text{S-Trans}\quad \frac{\tau_0 \leqslant: \tau_0' \qquad (\!|\, l_1\colon \tau_1, \ldots, l_m\colon \tau_m \,|\!) \leqslant: (\!|\, l_1'\colon \tau_1', \ldots, l_n\colon \tau_n' \,|\!)}{(\!|\, l_0\colon \tau_0, l_1\colon \tau_1, \ldots, l_m\colon \tau_m \,|\!) \leqslant: (\!|\, l_0\colon \tau_0', l_1'\colon \tau_1', \ldots, l_n\colon \tau_n' \,|\!)}$$

$\boxed{\Gamma \vdash e \rightsquigarrow e \cup a : \tau,\, v^{\mathcal{R}}}$

$$\text{T-Str}\quad \frac{s \in \mathbb{C}^*}{\Gamma \vdash s \rightsquigarrow s : \mathsf{String},\, \bot} \qquad\qquad \text{T-None}\quad \Gamma \vdash \mathsf{None} \rightsquigarrow \mathsf{None} : \mathsf{Maybe}\ \tau,\, \top$$

$$\text{T-Some}\quad \frac{\Gamma \vdash v \rightsquigarrow v' : \tau,\, v^{\mathcal{R}}}{\Gamma \vdash \mathsf{Some}\, v \rightsquigarrow \mathsf{Some}\, v' : \mathsf{Maybe}\ \tau,\, \top}$$

$$\text{T-Dict}\quad \frac{\Gamma \vdash v_0 \rightsquigarrow v_0' : \tau_0,\, v_1^{\mathcal{R}} \qquad \ldots \qquad \Gamma \vdash v_{n-1} \rightsquigarrow v_{n-1}' : \tau_{n-1},\, v_2^{\mathcal{R}}}{\begin{array}{l}\Gamma \vdash \{l_0 = v_0, \ldots, l_{n-1} = v_{n-1}\} \rightsquigarrow \{l_0 = v_0', \ldots, l_{n-1} = v_{n-1}'\} \\ \quad : (\!|\, l_0\colon \tau_0, \ldots, l_{n-1}\colon \tau_{n-1} \,|\!),\, \top\end{array}}$$

$$\text{T-Rx}^{=}\quad \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \mathsf{String},\, v^{\mathcal{R}} \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \mathsf{String},\, v^{\mathcal{R}}}{\Gamma \vdash \texttt{rx-match}\ e_1\ e_2 \rightsquigarrow \texttt{rx-match}\ e_1'\ e_2' : \mathsf{Maybe}\, (\!|\, l^{\#}\colon \mathsf{String} \,|\!),\, \top}$$

Figure 5: A simply typed language of regular expressions

Runtime Syntax (extends regexp syntax)

$$a \;=\; e \mid \mathsf{RegExp\text{-}Exn}$$
$$E \;=\; [\cdot] \mid E\ e \mid (\lambda\, x\,.\, e)\ E \mid \texttt{rx-match}\ E\ e \mid \texttt{rx-match}\ v\ E$$

$\boxed{e \longrightarrow e \cup a}$

$$\text{E-RxErr}\quad \frac{p \neq \alpha\beta\alpha\cdots\beta\alpha}{E[\texttt{rx-match}\ p\ s] \longrightarrow \mathsf{RegExp\text{-}Exn}} \qquad \text{E-RxMiss}\quad \frac{\mathcal{S}(s) \cap \mathcal{L}(p) = \varnothing}{E[\texttt{rx-match}\ p\ s] \longrightarrow E[\mathsf{None}]}$$

$$\text{E-RxMatch}\quad \frac{\begin{array}{c} s' = \mathsf{sup}(\mathcal{S}(s) \cap \mathcal{L}(p)) \\ p = \alpha_0\beta_1\alpha_1\cdots\beta_n\alpha_n \qquad (s_0, s'_1, s_1, \ldots, s'_n, s_n) = \mathsf{sup}(\mathcal{C}^{2n+1}(s')) \\ s_0 \in \mathcal{L}(\alpha_0) \\ \forall\, i \in \{\,1\,..\,n\,\}\,.\ s_i \in \mathcal{L}(\alpha_i) \wedge s'_i \in \mathcal{L}(\beta_i) \qquad \text{where } l_1, \ldots, l_n \text{ are the labels in } \beta_1, \ldots, \beta_n \end{array}}{E[\texttt{rx-match}\ p\ s] \longrightarrow E[\mathsf{Some}\ \{l^{\#} = s',\ l_1 = s'_1, \ldots,\ l_n = s'_n\}]}$$

Figure 6: The semantics of regular expressions

The output of a successful `rx-match` is a record with at least the designated field $l^{\#}$ bound to the matched substring s'. If the pattern contains any capturing sub-patterns $\beta_1, \ldots \beta_n$, the record has fields labeled $l_1, \ldots, l_n$ bound to matching substrings of s'. For a failed match, `rx-match` yields None.

In a simply typed language, the most precise type for the output of `rx-match` is an optional record with the single field $l^{\#}$. After all, it is impossible to predict the number of groups in p from the premise $\Gamma \vdash p : \mathsf{String}$.

4.2 Type-Tailored Regular Expressions

Figure 7 shows how type tailoring can refine the type of `rx-match` using the number of groups in the pattern string. The rule T-Rx^{-} detects malformed pattern strings, and the rule T-Rx^{+} specializes the result type to match the number of parenthesized groups apparent in the (well-formed) pattern. Assuming the elaboration system in figure 5 is sound, adding the rules in figure 7 yields a sound elaboration system.

Definition (*evaluation*): e *eval* a iff e is closed and there exists a term e' such that $\cdot \vdash e \rightsquigarrow e' : \tau$, $v^{\mathcal{R}}$ and $e' \longrightarrow^{*} a$.

Theorem (*soundness*): For all programs e, one of the following holds

- there exists a value v such that e *eval* v
- e *eval* RegExp-Exn
- *eval* is undefined for e.

$$\textsc{T-Pat}^{+}\ \frac{s \in \mathbb{C}^{*} \qquad s = \alpha_0\beta_1\alpha_1 \ldots \beta_n\alpha_n \quad \text{where } l_1, \ldots, l_n \text{ are the labels in } \beta_1, \ldots, \beta_n}{\Gamma \vdash s \rightsquigarrow s : \mathsf{String}, \langle l_1, \ldots, l_n \rangle} \qquad \textsc{T-Pat}^{-}\ \frac{s \in \mathbb{C}^{*} \qquad s \neq \alpha_0\beta_1\alpha_1 \ldots \beta_n\alpha_n}{\Gamma \vdash s \rightsquigarrow s : \mathsf{String}, \top}$$

$$\textsc{T-Rx}^{-}\ \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \mathsf{String}, \top \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \mathsf{String}, v^{\mathcal{R}}}{\Gamma \vdash \texttt{rx-match}\ e_1\ e_2 \rightsquigarrow \mathsf{RegExp\text{-}Exn} : \mathsf{Maybe}\ (\!|\ l^{\#} : \mathsf{String}\ |\!), \top}$$

$$\textsc{T-Rx}^{+}\ \frac{\Gamma \vdash e_1 \rightsquigarrow e_1' : \mathsf{String}, \langle l_1, \ldots, l_n \rangle \qquad \Gamma \vdash e_2 \rightsquigarrow e_2' : \mathsf{String}, v^{\mathcal{R}}}{\begin{array}{l}\Gamma \vdash \texttt{rx-match}\ e_1\ e_2 \rightsquigarrow \texttt{rx-match}\ e_1'\ e_2' \\ \quad : \mathsf{Maybe}\ (\!|\ l^{\#} : \mathsf{String}, l_1 : \mathsf{String}, \ldots, l_n : \mathsf{String}\ |\!), \top\end{array}}$$

Figure 7: Type-tailored regular expressions

Proof: T-Rx^{+} elaborates `rx-match` $e_1\ e_2$ to `rx-match` $e_1'\ e_2'$ at type:

$$\tau^{+} = \mathsf{Maybe}\ (\!|\ l^{\#} : \mathsf{String}, l_1 : \mathsf{String}, \ldots, l_n : \mathsf{String}\ |\!)$$

when e_1' is a well-formed pattern string containing labels $l_1, \ldots, l_n$, corresponding to n capture groups. Terms produced by T-Rx^{+} therefore step by either E-RxMiss or E-RxMatch, establishing progress.

Suppose now that (for some context E') $E'[\texttt{rx-match}\ e_1'\ e_2']$ steps to $E'[v]$. If the step is by E-RxMiss, then v is None and $\cdot \vdash v \rightsquigarrow v : \tau^{+}$ follows. Else the step is by E-RxMatch and because of the labels in e_1', the result v is:

$$\mathsf{Some}\ \{l^{\#} = s',\ l_1 = s_1', \ldots,\ l_n = s_n'\}$$

Again, $\cdot \vdash v \rightsquigarrow v : \tau^{+}$ follows. Furthermore τ^{+} is a subtype of $\mathsf{Maybe}\ (\!|\ l^{\#} : \mathsf{String}\ |\!)$; therefore, the hole type is preserved whether the elaboration of $E[\texttt{rx-match}\ e_1\ e_2]$ to $E'[\texttt{rx-match}\ e_1'\ e_2']$ proceeds by T-Rx^{+} or T-Rx$^{=}$. □

The key to soundness is again that the type elaborator identifies a run-time invariant (the pattern p has exactly n capture groups).

5 An API for a Type Elaborator: The Blueprint

Type tailoring requires an application programming interface to the type elaborator, that is, an API for meta programming. Based on the two introductory examples, such a meta-API must come with several capabilities: articulating new typing rules, inspecting values, computing during type checking, and altering the generated AST. This section formulate the desiderata for such an meta-API as a blueprint; the next section sketches how to translate this blueprint into a prototype implementation for Typed Racket.

$$\phi = [\kappa \rightarrow_f v^\kappa] \qquad v^{\mathcal{A}} = \bot \mid \mathbb{N} \mid \top$$
$$\kappa = \mathcal{A} \mid \mathcal{V} \mid \mathcal{R} \mid \mathcal{I} \mid \ldots \qquad v^{\mathcal{V}} = \bot \mid \mathbb{N} \mid \top$$
$$\Phi = \cdot \mid x : \phi, \Phi \qquad v^{\mathcal{R}} = \bot \mid \langle l_0, \ldots, l_{n-1} \rangle \mid \top$$
$$v^{\mathcal{I}} = \bot \mid \mathbb{Z} \mid \top$$

Figure 8: Proposition environment and value domains

Both preceding examples suggest at least three desiderata; any attempt to generalize the sample tailorings suggest a fourth one. First, the meta-API must obviously allow programmers to articulate type checking rules that exploit propositions about the values of an operation and its context. For example, the author of a "refinement" rule such as T-Rx$^+$ exploits a proposition about the shape of a string.

Second, a typing rule may have to use propositions from the expression's context, not just propositions about the expression itself. Hence, the type judgment synthesizes not only a type τ but also a proposition ϕ.

Third, the typing rules not only synthesize, they also propagate, propositions. For example, an ML-style `let` expression must propagate any proposition about pattern strings from the right-hand side of a variable declaration to the left-hand side and from there to the body of the `let` expression itself. In terms of the type judgment, this requirement means that mapping from variables to propositions must show up on the left side of the turnstile.

Fourth, different uses of the meta-API will synthesize and use different kinds of propositions about expressions. For example, the regular expression library relies on propositions about strings, while the array library collects propositions about index values. The implication for the meta-API is clear:

Each use of the meta-API owns a distinct slice of the synthesized propositions.

Concretely, the judgment therefore has to be of this shape:

$$\Phi; \Gamma \vdash e \rightsquigarrow e' : \tau,\ \phi$$

where Φ is an environment of proposition environments, Γ is a conventional type environment, and the term elaborates to a type τ and proposition environment ϕ. Thus the judgment states that in the context of Φ and Γ, the term e elaborates to e', its type is τ, and ϕ is a map of propositions that holds for e'.

A *proposition environment* maps variables to propositions (that index data by keys). The precise definition is on the left-hand side of figure 8. For concreteness, the figure includes four concrete keys that correspond to four abstract domains [4]. The sample domains are listed on the right side of the figure: procedure arities ($v^{\mathcal{A}}$), array lengths ($v^{\mathcal{V}}$), regular expression groups ($v^{\mathcal{R}}$), and integer constants ($v^{\mathcal{I}}$). The reader ought to imagine four distinct uses of the meta-API, each dealing with one of the domains and each having allocated a unique key. Uses can build on one another by importing the keys from other libraries.

$\boxed{\Phi; \Gamma \vdash e \rightsquigarrow e \cup a : \tau,\ \phi}$

P-VAR

$$\frac{\Gamma(x) = \tau \qquad \Phi(x) = \phi}{\Phi; \Gamma \vdash x \rightsquigarrow x : \tau,\ \phi}$$

P-LET

$$\frac{\Phi; \Gamma \vdash e_1 \rightsquigarrow e_1' : \tau_1,\ \phi_1 \qquad x : \phi_1, \Phi; x : \tau_1, \Gamma \vdash e_2 \rightsquigarrow e_2' : \tau_2,\ \phi_2}{\Phi; \Gamma \vdash \texttt{let}\, x = e_1\, \texttt{in}\, e_2 \rightsquigarrow \texttt{let}\, x = e_1'\, \texttt{in}\, e_2' : \tau_2,\ \phi_2}$$

P-LAM

$$\frac{x : \bullet, \Phi; x : \tau, \Gamma \vdash e \rightsquigarrow e' : \tau',\ \phi}{\Phi; \Gamma \vdash \lambda\, x : \tau\, .\, e \rightsquigarrow \lambda\, x\, .\, e' : \tau \to \tau',\ \bullet[\mathcal{A} \to_f 1]}$$

Figure 9: Generic elaboration rules

Conventions The notation $\phi[\kappa \to_f v^\kappa]$ denotes a finite map in which the key κ is bound to the value v^κ. The empty proposition map $\bullet$ binds each κ to its respective $\bot$. Also, $\Phi(x) = \bullet$ if x is not bound in Φ.

Some type elaboration rules remain generic. For example, the rules in figure 9 deal with variables, `let` expressions, and anonymous functions in a straightforward manner. For example, the type elaboration for `let` captures a map ϕ and retains it for later elaboration steps. Similarly, the elaboration rule for P-LAM invalidates any entry for x because propositions about a variable x do not hold in the body of the function.

The rules for specific applications of the meta-API must provide rules for their specific domains. For example, the use of the meta-API for array indexes may wish to supplement the rules for arithmetic—indexed to this particular domain. Similarly, the tailorings for regular expressions may want to leverage rules on string arithmetic, which are useful if regular expression patterns are created dynamically.

Finally, once these propositions are synthesized by the arguments to functions, "refinement" rules, such as the ones seen in the preceding sections, can exploit the propositions to improve type judgments.

In sum, a meta-API provides the framework for supplementing "simple" (as in non-dependent) type system with lightweight static analyses—and perhaps heavyweight ones in the future—that can assist programmers with the verification of simple type-like facts about their programs. The next section explains how to implement this idea in a full-fledged programming language.

6 An API for a Type Elaborator: The Prototype

To host a type elaborator API, a typed language must provide three core services:

- allow modifications of the program AST *before* type checking occurs;
- allow application programs to control the order of elaboration;
- provide a hook for attaching metadata to, an retrieving it from, AST nodes.

With the first two requirements, the implementor of a typing rule can implement the order of elaboration implied by the typing rules. The last requirement enables the modular implementation of the proposition maps.

Many existing typed languages meet these requirements. GHC,[9] Rust,[10] and Scala[11] offer plugin APIs, with which application programmers can create entirely new compiler passes. A programmer would need to build the "elaboration pass" from scratch, but has great freedom to do so.

In contrast, Typed Racket has a mature meta-API for the underlying compiler [5, 9]—because its implementation already relies on compiler APIs [18].[12] Specifically, Typed Racket's front end is a suite of syntax transformers that re-interpret Racket's syntax. The re-interpretation mostly inserts type information during the elaboration of the surface syntax into core syntax. Once a module is completely expanded, an ordinary type-checking pass checks the core language.

The meta-API for the type checker thus consists of three pieces:

- the meta-API for the compiler,
- a library for managing proposition environments (Φ`.rkt`);
- infrastructure for generic typing rules.

With this arrangement, the prototype implementation of P-REF^{+}, P-REF^{-}, and P-REF$^{=}$ consists of approximately twelve lines of code, show in figure 10.

The following subsections describe details of the implementation (section 6.1), the client interface to new tailorings (section 6.2), and the results from an evaluation of the usefulness of the tailorings described in sections 3 and 4 (section 6.3).

6.1 Constructing a Type Tailoring

The implementation of a type tailoring rule is just a syntax transformer. It consumes the equivalent of a node in a conventional abstract syntax tree and produces another such node. The key is that the result can be subjected to further such transformations.

Let us look at the the `tailored-vector-ref` syntax extension from figure 10, which illustrates the implementation of three elaboration rules. The syntax transformer consumes a fragment of the program such as:

```
(tailored-vector-ref '#(A B) 5)
```

The definition of `tailored-vector-ref` uses `syntax-parse` [5], a pattern-matching construct for abstract syntax nodes. It matches its argument and specifies what kind of code to generate if matching succeeds. In general, a `syntax-parse` clause has three parts:

- a parsing *pattern*;

[9] `https://downloads.haskell.org/~ghc/7.4.2/docs/html/users_guide/compiler-plugins.html`

[10] `https://doc.rust-lang.org/book/compiler-plugins.html`

[11] `http://www.scala-lang.org/old/node/140`

[12] Typed Clojure has a similar architecture.

tailored-vector.rkt

```
#lang typed/racket

; SERVICES:
(provide (rename-out [tailored-vector-ref vector-ref]
                     [tailored-vector-append vector-append]))
; 'vector-ref' is Racket's primitive for array dereferencing

; DEPENDENCIES:
(require racket/unsafe/ops)
(require (for-syntax racket/base))
(require (for-syntax syntax/parse))
(require
  (only-in (for-syntax "Φ.rkt")
           ~> I V ⊢))

; IMPLEMENTATION:
(define-syntax (tailored-vector-ref stx)
  (syntax-parse stx
   [(tailored-vector-ref e1:~> e2:~>)
    (define n (ϕ #'e1.~> V))
    (define i (ϕ #'e2.~> I))
    (cond
     [(not (and (integer? n) (integer? i)))
      #`(vector-ref e1.~> e2.~>)] ; P-Ref=
     [(and (<= 0 i) (< i n))
      #`(unsafe-vector-ref e1.~> e2.~>)] ; P-Ref+
     [else ; (or (< i 0) (>= i n))
      (error 'Index-Exn)])])) ; P-Ref-

(define-syntax (tailored-vector-append stx)
  (syntax-parse stx
   [(tailored-vector-ref e1:~> e2:~>)
    (define n1 (ϕ #'e1.~> V))
    (define n2 (ϕ #'e2.~> V))
    (define stx+ #'(vector-append #'e1.~> #'e2.~>))
    (cond
     [(and (integer? n1) (integer? n2))
      (⊢ stx+ (ϕ-update empty-ϕ V (+ n1 n2)))]
     [else
      stx+])]))
```

Figure 10: Implementing the type tailoring for array access

– a (possibly empty) series of actions on the surface syntax; and
– the *template*, which outlines the to-be-generated code.

If the input `stx` matches a clause's pattern, the Racket compiler creates a substitution environment from the free identifiers in the pattern (*pattern variables*). For example, the pattern variables in the (single) clause for `tailored-vector-ref` are `e1` and `e2`.

A pattern variable typically comes with an annotation, a suffix joined to the pattern variable with a colon (`:`). For example, `e:expr` says `e` is the name of the pattern variable and demands that it matches an expression; similarly, `e:integer` says `e` is a pattern variable and matches only integers. Programmers may implement their own annotations—so-called syntax classes—and the meta-API makes use of this capability by adding `~>` to the suite of annotations (imported from the implementation of proposition environments, `"Φ.rkt"`).

The actions on surface syntax can take the form of arbitrary Racket code. In brief, the tailoring for `vector-ref` checks whether the integer value `e2` is within the bounds of `e1` and conditionally produce an unsafe reference or an error term. (The elaborations of `e1` and `e2` infer these values and the function ϕ retrieves the values associated with the keys `V` and `I`.)

Lastly, an S-expression preceded by `#`` (pronounced "syntax quote") is a *code template*. Racket uses the substitution environment from the successful pattern match to replace the pattern variables in the code template with their current values. Thus, `#`(unsafe-vector-ref e1.~> e2.~>)` splices the *expanded form*[13] of the terms `e1` and `e2` into a program fragment that calls `unsafe-vector-ref`. Every path through a `syntax-parse` must produce a program fragment or an error term, and in fact the tailoring either produces an `unsafe-vector-ref`, produces an ordinary `vector-ref`,[14] or halts the compiler (with an `Index-Exn` or parse error).

The `tailored-vector-append` rule in figure 10 demonstrates an elaboration that records a new proposition. Given two arrays with known length, the `append` rule uses the $\vdash$ function to associate the sum of these lengths with the resulting syntax object. More precisely, $\vdash$ attaches a proposition map to a syntax object using Racket's *syntax properties*. Future elaborations can retrieve this stored value using the ϕ function.

6.2 Programming with Type Tailorings

To make the `tailored-vector-ref` tailoring available to client modules, it suffices to `provide` it. The second line of `tailored-vector.rkt` performs this export and additionally renames `tailored-vector-ref` to `vector-ref` for use in client modules. Here is one example use:

`client.rkt`

```
#lang typed/racket
(require "tailored-vector.rkt")
(vector-ref (vector-append '#(A) '#(B C)) 5)
```

[13] Implementation detail: when a piece of syntax matches the annotated pattern `e:~>`, later uses of the pattern variable `e` can access an *attribute* via `#`e.~>`. The definition of `~>` ensures that this attribute is bound the result of fully elaborating `e`.

[14] The full implementation also handles higher-order uses of `vector-ref`.

Compiling this module triggers an `Index-Exn` that points to the invalid access in the second line of the client module. Similarly, compiling a provably-valid access generates code for an unsafe reference that skips the dynamic bounds check.

The motivation for exporting `tailored-vector-ref` under the name `vector-ref` is so client modules can import the tailoring without refactoring any code. The tailored `vector-ref` shadows all former uses of `vector-ref` with nearly identical behavior. If the tailoring can infer the validity of an array reference, the client stands to benefit. Otherwise, the tailoring elaborates to an ordinary `vector-ref` and the client module functions exactly as before. In contrast, replacing `vector-ref` with a dependently typed function would guarantee that all array references are in bounds but also impose a *burden of proof* on all such references.

The API documentation for a library such as `tailored-vector.rkt` must communicate the logic that `tailored-vector-ref` uses to extend the standard `vector-ref`. One way to express this behavior is through the inference rules in figure 4. Another way is through normative language, for example:[15]

```
(vector-ref v i) → A
  v : [Vectorof A]
  i : Integer
```

> Returns the element in slot `i` of vector `v`. When elaboration can determine the length of `v` and value of `i`, `(vector-ref v i)` elaborates to an unsafe reference if `i` is a valid slot and raises `Index-Exn` otherwise.

This specification conveys the type of `vector-ref` in formal notation and the informal guarantee of type tailoring in plain language. Details about "when elaboration can determine" such propositions belong in a general section of the documentation.

6.3 Evaluation

Figure 11 presents the results of applying type tailoring for array references and regular expression matching to existing code. Specifically, the prototype implementation tracks vector bounds (as described in section 3) and parses POSIX regular expressions (as opposed to the simplified notation in section 4) to determine grouping information. The prototype furthermore associates propositions to variables bound by Typed Racket's `let` and `define` forms in the manner described by figure 9. Consequently, the implementation cannot validate simple vector references such as:

```
((λ (x) (vector-ref x 0)) '#(A B C))
```

Nevertheless, figure 11 demonstrates that real programs use vector constants (occasionally) and regular expression literals (frequently). For example, an implementation of `gzip` in Racket declares array constants to implement a Huffman tree and heap.

The scope of the evaluation is the Racket 6.5 core distribution and third-party packages.[16] Many of these files are untyped, but the elaborations still apply.

[15] `http://docs.racket-lang.org/trivial/index.html`

[16] `http://pkgn.racket-lang.org/`

Files Searched	10,131		
Files using `vector-ref`	64	Files using `regexp-match`	161
Files optimized	7	Files refined	123
Total Num. `vector-ref`	385	Total Num. `regexp-match`	323
Num. `vector-ref` optimized	97	Num. `regexp-match` refined	263
		Type errors fixed	327

Figure 11: Evaluation summary

Regarding `vector-ref`, only a handful of modules use the operation and only seven such modules dereference vectors in a way that the elaboration rules can validate. In terms of raw `vector-ref` sites in these modules, however, 30% of all occurrences benefit from the tailored rules. Many of these occurrences (80 successes) are from a module implementing a board game that uses an inlining macro to unroll a loop over all pawns on the board to straight-line references at constant offsets. The other successes optimize reads to constant offsets in fixed-size buffers.

On one hand, this result is not surprising. Functional (Racket) programmers do not immediately think of arrays when they represent data but resort to other forms of data structures instead. They switch to arrays only when performance becomes important. On the other hand, it is still encouraging that the evaluation yields a fair number of improvements for this type tailoring.

The results for `regexp-match` are much stronger. While only 161 files use regular expression patterns with subgroups, almost all such uses benefit from the type tailoring (over 80%). Furthermore, type tailoring resolves 327 false "type errors" reported by Typed Racket. This total is the sum of type errors reported by Typed Racket *without tailoring* on two classes of contexts using `regexp-match`:

- 117 originally-untyped contexts, ported to Typed Racket *with tailoring*
- 6 originally-typed contexts, after removing casts made redundant by tailoring

The reason for redundant assertions is that Typed Racket must give a conservative type to `regexp-match`. This is because subgroups may fail to capture even though the overall match succeeds. For example:

```
> (regexp-match "(a)|(b)" "a")
'("a" "a" #f)
```

A type that accomodates this behavior is:[17]

```
(regexp-match pat str)
 → (Option (Pairof String (Listof (Option String))))
  pat : String
  str : String
```

[17] The actual type allows byte string and port inputs.

Therefore most calls to `regexp-match` in typed code must explicitly check that the groups actually captured a substring, even when capture is guaranteed.

The remaining 38 files that are not improved by tailoring either use non-constant pattern strings (5 files), compute the pattern with a function like `string-append` (9 files), have groups that may not capture (4 files), or use the extracted groups in code that does not care if the result was a string or `#false`, e.g., in a call to `printf` (20 files).

7 Inspiring and Related Work

Herman and Meunier [14] demonstrate how macro systems can implement essential elements of partial evaluation. They show how this benefits the static analysis of `printf` format strings, regular expression matching, and SQL database queries.[18] Their work, which pre-dates Typed Racket, serves as inspiration. From far enough away, this paper turns their idea into a systematic approach and integrates it with type checking. The details of their implementation and the one for type tailoring differ drastically.

A second piece of inspiration is due to version 7.10 of the Glasgow Haskell Compiler. It comes with an API for the type checker, specifically the type constraint solver.[19] A client programmer may register external solvers via the GHC type constraint API so that when the default solver cannot solve a set of constraints, it ships the constraints to the user-defined solver. If a new solver can make any progress, it returns a modified list of constraints along with proofs that its solutions are valid. Three user-defined constraint solvers are known: a plugin for solving natural number constraints by normalization,[20] a plugin for solving numeric constraints via SMT [6], and an algebra for units of measure [11]. All three are powerful and useful extensions. The proofs they submit to GHC are invariably "proofs by assertion."

Two related pieces of work concern language extension and language modeling. Fisher and Shivers [8] present *Ziggurat*, a macro system for C-like languages, inspired by Scheme and implemented with Scheme syntax. This macro system allows programmers to attach arbitrary analysis computations to macros. Although Fisher and Shivers do not apply this idea to extend the underlying type system, it ought to be possible to use this system to implement some form of type tailoring.

SoundX is a system for modeling programming languages and type-sound extensions [15]. A programmer may define typed syntax for a `let` statement in terms of a pre-defined typing rule for λ expressions. SoundX typechecks this extension itself, providing early and actionable feedback regarding potential type errors. Thus, desugarings are guaranteed to produce only well-typed terms. While the SoundX approach has some similarity to an elaborator API, the two differ radically because Typed Racket checks types only after *all* syntax has been elaborated. As a result, it is possible to interpose between the SoundX parser and desugarer.

Finally, type tailoring adds a flavor of dependent typing to simple type systems, which calls for some brief comparison. For example, the `printf` and `vector-ref` elaborations are similar to those found in the literature on dependent types [19]. The move

[18] In fact, our prototype tailors `printf` and the `db/postgres` library.

[19] `ghc.haskell.org/trac/ghc/wiki/Plugins/TypeChecker`

[20] `http://christiaanb.github.io/posts/type-checker-plugin/`

from “simple” type system to dependent type systems comes with an unexcelled payoff in terms of expressing and enforcing “deep” correctness specifications. The price for this additional expressive power is high, however [1, 16]. Programmers must not only design functions systematically—an already difficult task—but they must also learn to design dependent types and proofs, and constructing those may introduce substantial development and maintenance overhead.

While type tailoring as introduced here is not a substitute for dependent types and does not even come close, the evaluations suggest that it injects a modicum of dependent typing into “simple” type systems—at no cost to the client programmer. Only the creator of libraries must use the type elaborator API. In other words, type tailoring introduces an additional element into the spectrum between “simple” and “advanced” type systems.

8 Things Take Time

A programming language changes slowly because it offer guarantees. Its type system, by necessity, changes even more slowly because the rest of the language depends on it. Programmers’ needs change more quickly than languages, however, and therefore useful type systems must be able to grow in ways that their designers could not imagine [12]. Type tailoring is a new and simple way to grow a type system.

The elaborator API for Typed Racket and its sample uses—printing, array indexing, regular expression matching plus related features in these libraries—scratches only the surface. Typed Racket comes with many libraries that deserve type tailoring: database queries, FFI, `xml` and `html`, graphics toolboxes, process spawning, and many more. Similarly, the libraries of many other programming languages—Haskell, Java, OCaml, Rust, or Scala—could benefit from type tailoring.

One challenge is to design APIs for the type elaborators of these languages. For example, Rust and Scala programmers may already program the compiler via plug-ins. In Rust’s case, the compiler API supports both Racket’s weak form of notational definition and functions on typed ASTs; Scala has even more mature plug-ins than Rust. The question is whether these extension systems can already implement elaborator APIs or whether they need access to more of the internals of the compiler.

Programming the compiler is dangerous yet language designers nevertheless support compiler plug-ins. They know that programmers need new language constructs. In the same spirit, programmers *ought* to be able to program type systems in the same direct manner as compilers. While giving access to the type checker is as dangerous as programming the compiler, it comes with huge potential, as this paper demonstrates with a relatively simple prototype.

References

[1] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving Acceptability Properties of Relaxed Nondeterministic Approximate Programs. In *Proc. ACM Conference on Programming Language Design and Implementation*, pp. 169–180, 2012.

[2] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally Tagless, Partially Evaluated. In *Proc. Journal Functional Programming*, pp. 509–543, 2009.

[3] Stephen Chang, Alex Knauth, and Ben Greenman. Macros as Types. In *Proc. ACM Symposium on Principles of Programming Languages*, 2017. To appear.

[4] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 238–252, 1977.

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

[6] Iavor Diatchki. Improving Haskell Types with SMT. In *Proc. ACM Symposium on Haskell*, pp. 1–10, 2015.

[7] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: a Programming Environment for Scheme. *Journal Functional Programming* 12(2), pp. 159–182, 2002.

[8] David Fisher and Olin Shivers. Building Language Towers with Ziggurat. *Journal Functional Programming* 18(6), pp. 707–780, 2008.

[9] 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.

[10] Emden R. Gansner and John H. Reppy. The Standard ML Base Library. 1St edition. Cambridge University Press, 2004.

[11] Adam Gundry. A Typechecker Plugin for Units of Measure: Domain-Specific Constraint Solving in GHC. In *Proc. ACM Symposium on Haskell*, pp. 11–22, 2015.

[12] Guy L. Steele, Jr. Growing a Language. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pp. 0.01–A1, 1998.

[13] Robert Harper and John C. Mitchell. On the Type Structure of Standard ML. *Transactions on Programming Languages and Systems* 15(2), pp. 211–252, 1993.

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

[15] Florian Lorenzen and Sebastian Erdweg. Sound Type-Dependent Syntactic Language Extension. In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 204–216, 2016.

[16] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified Peephole Optimizations for CompCert. In *Proc. ACM Conference on Programming Language Design and Implementation*, pp. 448–461, 2016.

[17] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation Validation. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 151–166, 1998.

[18] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as Libraries. In *Proc. ACM Conference on Programming Language Design and Implementation*, pp. 132–141, 2011.

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

9 Appendix

Proofs of theorems in section 3

Theorem (*soundness*): For all programs e, one of the following holds:
- there exists an integer v such that e *eval* v
- e *eval* Index-Exn
- *eval* is undefined for e.

Proof: The proof uses a variant of the progress and preservation strategy. Usually a preservation lemma proves that a rewriting step of one program to another preserves types (or yields an exception). Because the type judgment for a program is a binary relation, simple tree surgery suffices to re-establish the type correctness proof for the rewritten program. In the setting of type elaboration, the proof must also show that there exists a surface term that elaborates to the rewritten program.
While establishing the proof for a bijective elaboration is straightforward, the idea deserves two comments. The first one develops the fundamental idea of how array indexing is executed in a type safe manner. The second one applies this idea to progress and preservation.

First, suppose $\Gamma \vdash E[e] \rightsquigarrow E'[e'] : \mathsf{Int},\ v_1^{\mathcal{V}}$ and $E'[e']$ steps to $E'[e'_{\rightarrow}]$. Then there exists a term $e_{\rightarrow}$ such that $\Gamma \vdash E[e_{\rightarrow}] \rightsquigarrow E'[e'_{\rightarrow}] : \mathsf{Int},\ v_2^{\mathcal{V}}$. The proof proceeds by cases on the evaluation rule showing $E'[e'] \rightarrow E'[e'_{\rightarrow}]$. Here are two:
- Case E-APP: e' is an application $(\lambda\, x\,.\, f')\ v'$, therefore $\Gamma \vdash v \rightsquigarrow v' : \tau,\ v_3^{\mathcal{V}}$ and $x : \tau, \Gamma \vdash f \rightsquigarrow f' : \tau',\ v_4^{\mathcal{V}}$ are premises from the type derivation for $E'[e']$. The result $e'_{\rightarrow}$ of taking a step is $f'[v'/x]$. By the substitution lemma below, $\Gamma \vdash f[v/x] \rightsquigarrow f'[v'/x] : \tau',\ v_4^{\mathcal{V}}$. With $f[v/x]$ as $e_{\rightarrow}$, the goal $\Gamma \vdash E[e_{\rightarrow}] \rightsquigarrow E'[e'_{\rightarrow}] : \mathsf{Int},\ v_2^{\mathcal{V}}$ follows.
- Case E-CHECK: e' is `checked-ref` $\langle v_0, \ldots, v_{n-1}\rangle\ i$ and $e'_{\rightarrow}$ is v_i. The type derivation of e' implies that $\langle v_0, \ldots, v_{n-1}\rangle$ is an array and E-CHECK implies $0 \leqslant i < n$; it follows that v_i is typed by T-INT.

Second, $\Gamma \vdash e \rightsquigarrow e' : \mathsf{Int},\ v_1^{\mathcal{V}}$ implies e' can make progress. Moreover if the result of this step is another term $e'_{\rightarrow}$, then the same premise suffices to establish the preservation property, that is, it can validate $\Gamma \vdash e_{\rightarrow} \rightsquigarrow e'_{\rightarrow} : \mathsf{Int},\ v_2^{\mathcal{V}}$ with an $e_{\rightarrow}$ derived from $e'_{\rightarrow}$. □

Lemma (*substitution*): If $x : \tau, \Gamma \vdash e \rightsquigarrow e' : \tau',\ v_1^{\mathcal{V}}$ and $\Gamma \vdash v \rightsquigarrow v' : \tau,\ v_2^{\mathcal{V}}$ then $\Gamma \vdash e[v/x] \rightsquigarrow e'[v'/x] : \tau',\ v_1^{\mathcal{V}}$

Proof: By induction on the derivation of $x : \tau, \Gamma \vdash e \rightsquigarrow e' : \tau',\ v_1^{\mathcal{V}}$.
- Case T-VAR: The goal is $\Gamma \vdash y[v/x] \rightsquigarrow y[v'/x] : \tau',\ v_1^{\mathcal{V}}$ for some variable y. If $y = x$ then $\tau = \tau'$. Furthermore, $v_1^{\mathcal{V}} = v_2^{\mathcal{V}} = \bot$ and the goal after substitution is the premise $\Gamma \vdash v \rightsquigarrow v' : \tau,\ \bot$. Otherwise, the goal is the premise $\Gamma \vdash y \rightsquigarrow y : \tau',\ v_1^{\mathcal{V}}$.
- Case T-INT: Immediate, because e and e' are the same integer n and therefore $e[v/x] = e'[v'/x] = n$.
- Case T-LAM: e and e' are functions $(\lambda\, y : \tau^f\,.\, f)$ and $(\lambda\, y\,.\, f')$. If $x = y$ the goal is immediate. Otherwise, it suffices to prove $y : \tau'', \Gamma \vdash f[v/x] \rightsquigarrow f'[v'/x] : \tau',\ v_1^{\mathcal{V}}$ which follows by the induction hypothesis.

– Cases T-ARR, T-APP, and T-REF$^{=}$ follow by the induction hypothesis applied to their subterms.

□

Theorem (*soundness, revised*): For all programs e, one of the following holds:
– there exists an integer v such that e *eval*^{u} v
– e *eval*^{u} Index-Exn
– *eval*^{u} is undefined for e.

Proof: Surface terms of the form $e_1@e_2$ can now elaborate by T-REF$^{=}$ or T-REF^{+} before they are evaluated. The case analysis of the original soundness proof covers the former. The latter adds a case.
Although bijectivity is gone, the *proof tree* that synthesizes a redex provides a trace back to the surface syntax. Part of the trace includes conditions that help establish progress and preservation for redexes involving `unsafe-ref`. Concretely, the elaborated redex is `unsafe-ref` $\langle v_0, \ldots, v_{n-1} \rangle\ i$ and, by the pre-conditions in the proof tree, $0 \leqslant i < n$.
Progress for T-REF^{+} obviously holds.
Preservation is re-established as follows. First, the redex `unsafe-ref` $\langle v_0, \ldots, v_{n-1} \rangle\ i$ steps to the well-typed integer v_i by the premise in the elaboration proof tree and E-UNSAFE. Second, the substitution lemma holds for terms derived via T-REF^{+} by the induction hypothesis applied to subterms $\langle v_0, \ldots, v_{n-1} \rangle$ and i. □

Implementation Sketch

Figure 10 presented tailorings for `vector-ref` and `vector-append` that depended on an implementation of the proposition maps, ϕ. Figure 12 demonstrates one such implementation using Racket's hashtables. In particular, a proposition map is a hashtable from symbolic keys to arbitrary values. Any client of this API can register a new class of key (κ in figure 8) by calling ϕ`-update`. Clients can retrieve keys using the ϕ function, which serves two purposes:

– Calling `(`ϕ `stx)` returns the proposition map for the AST node `stx`, or a new map if one did not already exist.
– Calling `(`ϕ `stx k)` is short for `(`ϕ`-ref (`ϕ `stx) k)`.[21]

With this interface, the rule implementing `(let ([x e1]) e2)` simply retrieves the proposition map for `e1` and binds it to `x` in the scope of `e2`. Racket's compiler API includes a form `make-rename-transformer` that handles this task automatically. Figure 13 demonstrates how the "map of maps" Φ comes for free in the Typed Racket prototype.

Finally, figure 14 tailors `regexp-match` assuming the regular expression syntax from section 4.1. The novelty of this implementation is that the function `refine-type` generates code for a fixed-length list using `list-ref`. Typed Racket trusts these list references for the same reason it cannot give a precise type for `regexp-match` in the first place — because the type system lacks dependent types.

[21] Figure 12 differs from the actual implementation in one crucial way. When a lookup fails in figure 12, the return value is `#false`. In the prototype, the return value is the $\bot$ associated with the current abstract domain.

Φ.rkt

```
#lang (provide (all-defined-out))

(define ϕ-key ; for storing syntax properties
  (gensym 'ϕ))

(define empty-ϕ
  (hasheq))

(define (ϕ-ref ϕ k)
  (hash-ref ϕ k))

(define (ϕ-update ϕ d v)
  (hash-set ϕ k v))

(define (ϕ stx [k #f])
  (define prop
    (or (syntax-property stx ϕ-key)
        (ϕ-init)))
  (if k
    (ϕ-ref prop k)
    prop))

(define (⊢ stx new-ϕ)
  (syntax-property stx ϕ-key new-ϕ))
```

Figure 12: Implementing a proposition environment

let.rkt

```
#lang (provide (rename-out [tailored-let let]))
(require "Φ.rkt")

(define-syntax (tailored-let stx)
  (syntax-parse stx
    [(let ([name e1:~>]) e2)
     #:with ϕ1 (ϕ #'e1.~>)
     (syntax/loc stx
       (let ([name e1.~>])
         (let-syntax ([name (make-rename-transformer
                             (⊢ #'name 'ϕ1))])
           e2)))]))
```

Figure 13: Implementing `let`

rxm.rkt

```
#lang typed/racket
(provide (rename-out [tailored-regexp-match regexp-match]))
(require (for-syntax racket/base syntax/parse)
         "Φ.rkt")


(define-syntax (tailored-regexp-match stx)
  (syntax-parse stx
    [(_ p:~> s)
     (define ng (ϕ #'p.~> R))
     (cond
       [(eq? ng ⊥)
        #'(regexp-match p.~> s)] ; T-Rx=
       [(not ng)
        (error "RegExp-Exn")] ; T-Rx-
       [else
        (refine-type #`p.~> #`s ng)])])) ; T-Rx+


;; Syntax Syntax Natural -> Syntax
(define-for-syntax (refine-type p s n)
  #`(let ([xs (regexp-match p s)])
      (if (not xs)
          #false
          (list
            (car xs)
            #,@(for/list ([i n])
                 #`(list-ref xs (+ #,i 1)))))))

;; Used internally, by the P-Pat tailorings
;; String Natural Boolean -> (Option Natural)
(define-for-syntax (num-groups str [i 0] [g #false])
  (cond
    [(= i (string-length str))
     (and (not g) 0)]
    [(eq? #\( (string-ref str i))
     (and (not g) (num-groups str (+ i 1) #true))]
    [(eq? #\) (string-ref str i))
     (define g+
       (and g (num-groups str (+ i 1) #false)))
     (and g+ (+ 1 g+))]
    [else
     (num-groups str (+ i 1) g)]))
```

Figure 14: Implementing `regexp-match`