Type Systems as Macros

Stephen Chang  Alex Knauth  Ben Greenman

PLT @ Northeastern University, Boston, MA, USA
{stchang,alexknauth,types}@ccs.neu.edu

Abstract

We present TURNSTILE, a metalanguage for creating typed embedded languages. To implement the type system, programmers write type checking rules resembling traditional judgment syntax. To implement the semantics, they incorporate elaborations into these rules. TURNSTILE critically depends on the idea of linguistic reuse. It exploits a macro system in a novel way to simultaneously type check and rewrite a surface program into a target language. Reusing a macro system also yields modular implementations whose rules may be mixed and matched to create other languages. Combined with typical compiler and runtime reuse, TURNSTILE produces performing typed embedded languages with little effort.

Categories and Subject Descriptors  D.3.2 [Programming Languages]: Specialized application languages

Keywords  macros, type systems, typed embedded DSLs

1. Typed Embedded Languages

As Paul Hudak asserted, “we really don’t want to build a programming language from scratch ... better, let’s inherit the infrastructure of some other language” [43]. Unsurprisingly, many modern languages support the creation of such embedded languages [3, 18, 20, 22, 24–26, 41, 43, 46].

Programmers who wish to create typed embedded languages, however, have more limited options. Such languages typically reuse their host’s type system but, as a prominent project [45] recently remarked, this “confines them to (that) type system.” Also, reusing a type system may not create proper abstractions, e.g., type errors may be reported in host language terms. At the other extreme, a programmer can implement a type system from scratch [42], expending considerable effort and passing up many of the reuse benefits that embedding a language promises in the first place.

We present an alternative approach to implementing typed embedded languages. Rather than reuse a type system, we embed a type system in a host’s macro system. In other words, type checking is computed as part of macro expansion. Such an embedding fits naturally since a typical type checking algorithm traverses a surface program, synthesizes information from it, and then uses this information to rewrite the program, if it satisfies certain conditions, into a target language. This kind of algorithm exactly matches the ideal use case for macros. From this perspective, a type checker resem-
blies a special instance of a macro system and our approach exploits synergies resulting from this insight.

With our macro-based approach, programmers may implement a wide range of type rules, yet they need not create a type system from scratch since they may reuse components of the macro system itself for type checking. Indeed, programmers need only supply their desired type rules in an intuitive mathematical form. Creating type systems with macros also fosters robust linguistic abstractions, e.g., they report type errors with surface language terms. Finally, our approach produces naturally modular type systems that dually serve as libraries of mixable and matchable type rules, enabling further linguistic reuse [27]. When combined with the typical reuse of the runtime that embedded languages enjoy, our approach inherits the performance of its host and thus produces practical typed languages with significantly reduced effort.

We use Racket [12, 13], a production-quality Lisp and Scheme descendant, as our host language since Lisps are already a popular platform for creating embedded languages [17, 20]. Racket’s macro system in particular continues to improve on its predecessors [13] and has even influenced macro system design in modern non-Lisp languages [6, 8, 19, 48]. Thus programmers have created Racket-embedded languages for accomplishing a variety of tasks such as book publishing [7], program synthesis [44], and writing secure shell scripts [32].

The first part of the paper [43] demonstrates a connection between type rules and macros by reusing Racket’s macro infrastructure for type checking in the creation of a typed embedded language. The second part [44] introduces TURNSTILE, a metalanguage that abstracts the insights and techniques from the first part into convenient linguistic constructs. The third part [51] shows that our approach both accommodates a variety of type systems and scales to realistic combinations of type system features. We demonstrate the former by implementing fifteen core languages ranging from simply-typed to $F_{\omega}$, and the latter with the creation of a full-sized ML-like functional language that also supports basic Haskell-style type classes.

2. Creating Embedded Languages in Racket

This section summarizes the creation of embedded languages with Racket. Racket is not a single language but rather an ecosystem with which to create languages [12]. Racket code is organized into modules, e.g. LAM[1]

```
@provide
(define-m (lm x e) (\x (x) e))
```

Code note: For clarity and conciseness, this paper stylizes code and thus its examples may not run exactly as presented. Full, runnable examples are available at: [www.ccs.neu.edu/home/stchang/popl2017](http://www.ccs.neu.edu/home/stchang/popl2017)

---

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM 978-1-4503-4660-3/17/01...
DOI: http://dx.doi.org/10.1145/3009837.3009886
A `#lang racket` declaration allows LAM to use forms and functions from the main Racket language. LAM defines and exports one macro `lm`, denoting single-argument functions. A Racket macro consumes and produces syntax object data structures. The `lm` macro specifies its usage shape with input pattern `(lm x e)` (in yellow to help readability), which binds pattern variables `x` and `e` to sub-pieces of the input, the parameter and body, respectively. The output syntax `(λ (x e) e)` (gray denotes syntax object construction) references these pattern variables `(λ` is Racket’s `λ`).

A module serves multiple roles in the Racket ecosystem. Running LAM as a program produces no result since it consists of only a macro definition. But LAM is also a language:

```
#lang racket
(lm x (lm y x)) ;=> (function)
((lm x x) (lm x x)); stx error! fn application undefined
```

A module declaring `#lang lam` may only write `lm` functions, using any other form results in an error. Finally, a Racket module may be used as a library, as in the following LC module:

```
#lang lc
(require lam)
(provide (rename-out [lm λ] [app #%app]))
(define-m (app e_fn e_arg) (#%app e_fn e_arg))
```

LC imports `lm` from LAM and also defines `app`, which corresponds to single-argument function application. LC exports `lm` and `app` with new names, `λ` and ` #%app`, respectively. The `%app` in the output of `app` is core-Racket’s function application form, though programmers need not write it explicitly. Instead, macro expansion implicitly inserts it before applied functions. This enables modifying the behavior of function application, as we do here by exporting `app` as `%app`. Thus a program in the LC language looks like:

```
#lang lc
((λ (x x)) (λ (x (x x))); => loop!
```

where `λ` corresponds to `lm` in LAM and applying a `λ` behaves according to `app` in LC. Running LC-PROG loops forever.

Figure 1 depicts compilation of a Racket program, which includes macro expansion. The Racket compiler first “reads” a program’s surface text into a syntax object, which is a tree of symbols and literals along with context information, e.g., in-scope bindings and source locations. The macro expander then expands macro invocations in this syntax object according to macro definitions from the program’s declared `#lang`. Macro expansion may reveal additional macro uses or even define new macros, so expansion repeats until no macro uses remain. Compilation terminates with a syntax error if expansion of any macro fails. The output of macro expansion contains only references to Racket’s core syntax. This paper shows how to embed type checking within macro expansion.

### 3. A Typed λ-Calculus Embedded Language

LC from section 2 implements the untyped λ-calculus. This section augments LC with types and type checking by transcribing formal type rules directly into its macro definitions, producing the simply-typed λ-calculus and demonstrating that Racket’s macro infrastructure can be reused for type checking. Figure 2 presents the standard simply-typed λ-calculus rules, and a skeleton implementation. The macros in this implementation also erase types in the surface language to accommodate the untyped Racket host.

#### 3.1 Typed Function Application

Figure 3 presents `checked-app`, a macro that elaborates typed function application nodes into core Racket and also checks the syntax tree (“v0” marks this initial version). Additional `#:with` and `#:when` conditions guard the macro’s expansion. A pattern and expression follow a `#:with` and macro expansion continues only if the result of evaluating the latter produces a syntax object that matches the former. The first `#:with` uses a compute-τ function to compute the type of function `e_fn`, which must match pattern `(→ τ_in τ_out)`. The second `#:with` computes the type of argument `e_arg`, binding it to pattern variable `τ_arg`. Unlike the first `#:with`, the `τ_arg` pattern does not constrain the shape of `e_arg`’s type but the following `#:when` asserts that `τ_arg` and `τ_in` satisfy predicate `τ`. The types in `e_fn` and `e_arg` are then erased (lines 5–6) before they are emitted in the macro’s output (overlines mark type-erased expressions, and core Racket forms). Finally, `add-τ` (line 7) “adds” `τ_out` to the macro’s syntax object output. In summary, `checked-app` rewrites a typed function application to an equivalent untyped one, along with its type.

---

**Figure 1.** The Racket compiler’s frontend

<table>
<thead>
<tr>
<th>Program surface text</th>
</tr>
</thead>
<tbody>
<tr>
<td>READER</td>
</tr>
<tr>
<td>Syntax object: surface language (with macro calls)</td>
</tr>
</tbody>
</table>

**Figure 2.** Simply-typed λ-calculus

| 1 (define-m (checked-app e_fn e_arg) ;v0 |
| 2 #:with (→ τ_in τ_out) (compute-τ e_fn) |
| 3 #:with τ_arg (compute-τ e_arg) |
| 4 #:when (τ_arg τ_in) |
| 5 #:with e_fn (erase-τ e_fn) |
| 6 #:with e_arg (erase-τ e_arg) |
| 7 (add-τ (%app e_fn e_arg) τ_out) |

**Figure 3.** A type checking function application macro
3.2 Communicating Macros

The organization of checked-app in figure 4 resembles a combination of its T-APP and erase specification in figure 2. Figure 4 completes checked-app by defining some helper functions, which together establish a communication protocol between type rule macros. These functions utilize syntax properties, which are arbitrary key-value pairs stored with a syntax object’s metadata. For example, checked-app calls add-τ to attach type information to its output, which in turn calls add-stx-prop (figure 4, line 1) to associate a type τ with key ‘type’ on expression e. If all type rule macros follow this protocol, then to compute an arbitrary expression’s type, we simply invoke that expression’s macro and retrieve the attached type from its output. In other words, expanding an expression also type checks it.

We can call Racket’s macro expander to invoke the desired type checking macro but not in the standard manner. Macro expansion typically rewrites all macro invocations in a program at once (figure 4) and repeats this process until there are no more macro calls. Such breadth-first expansion is incompatible with type checking, however, which proceeds in a depth-first manner—a term is well-typed only if its subterms are well-typed—but the local-expand function controls expansion in the desired way, expanding just one syntax object without considering other parts of the program. Thus compute-τ expands its argument with local-expand (figure 4, line 3) and then retrieves its type.

The checked-app macro uses erase-τ to produce syntax without type annotations. If all type rule macros follow this protocol then expanding an expression also erases its types. Separate calls to compute-τ and erase-τ, however, unnecessarily expands syntax twice. The comp+erase-τ function (lines 5-7) eliminates this redundancy and figures 5’s revised checked-app uses this function. In general, we carefully avoid extraneous expansions while type checking so as not to change the algorithmic complexity of macro expansion.

Finally, type checking requires a notion of type equality. We cannot compute mere symbolic equality since types are renamable linguistic constructs:

\[(\text{require } (\text{rename } \rightarrow a)]) \quad (\tau \equiv (a \to t) \to (s \to t)) \quad \Rightarrow \text{true}\]

If we represent types with syntax objects, however, type equality is syntax equality and we can reuse Racket’s knowledge of the program’s binding structure (stx= in figure 4, line 8) to compute type equality in a straightforward manner.

3.3 Type Environments and Type Checking λ

Figure 6 implements the → type (lines 1-2) as a macro that matches on an input and output type and expands to an application of an internal function that errors at runtime (there are no base types for now, see §3.4). The checked-λ macro requires a type annotation on its parameter (line 4), separated with :. This macro resembles checked-app, except a new comp+erase-τ/ctx function replaces comp+erase-τ. Since the λ body may reference x, comp+erase-τ/ctx computes the body’s type in a type context containing x and its type, given as the second argument.

So far, checked-app and checked-λ correspond to T-APP and T-ABS from figure 2, respectively. To implement T-VAR, i.e., type environments, comp+erase-τ/ctx defines a local macro with let-macro (figure 6, line 12) and expands an expression e in the scope of this new macro. The local macro is named x and expands to a fresh y that has the desired type τ attached (observe the nested gray highlights). As a result, while expanding e, a reference to x (with no type information) becomes a reference to y (with type information). To avoid undefined errors during expansion, a (Racket) λ wraps the let-macro before expansion. Finally, comp+erase-τ/ctx returns a tuple of post-expansion y (as τ), the type-erased x, and its type τout. Effectively, defining a local macro inserts a binding indirection level during macro expansion, enabling the insertion of the desired type information on variable references. Thus T-VAR is implemented, reusing the compile-time macro environment as the type environment. This completes our simply-typed language.

3.4 A Few Practical Matters

We have implemented a basic λ-calculus; however, we wish to implement practical languages. This subsection shows how to extend our language with features found in such languages.

Multiple arguments Figure 7 revises our simply-typed language to support multiple arguments. An ellipsis pattern (...) matches zero-or-more of the preceding element. If that preceding element

\[\text{let-macro abbreviates Racket’s let-syntax and syntax-parse.}\]
binds pattern variables, ellipses must follow later references to those variables, e.g., the revised \( \to \) macro (line 1) matches zero-or-more input arguments \( \tau_{\text{in}} \) and ellipses follow \( \tau_{\text{in}} \) in its output. The other forms are extended similarly. The \( \text{checked-} \lambda \) macro uses a slightly modified \( \text{comp+erase-} \tau \) (line 5) that accepts multi-element contexts. In \( \text{checked-app} \) (line 5), the “vector” \( \vec{\tau} \) notation denotes \( f \) mapped over its input list.

**Error messages** Figure 7 also reports more useful error messages. The \( \text{checked-app} \) in figure 8 reports type errors as syntax errors but a better message should indicate the error’s location and the computed and expected types. The \( \text{checked-app} \) in figure 7 reports such a message using a \( \#\text{fail-unless} \) condition (lines 6-8) to produce a message from a printf-style format string (\( \text{this-stx} \) is the current input syntax, analogous to the OO “this”). All our languages strive to report accurate messages in the manner of figure 7, though the paper may not always show this code.

**Type well-formedness** Our language so far checks the types of terms but does not check whether programmer-written types are valid, e.g., \( (\lambda (\vec{x} : \to)) \) \( x \) or \( (\lambda (\vec{x} : \text{Undef}) \) \( x \) are valid programs according to figure 7. Applying these functions result in type errors but the invalid types should be reported before then. Many type checkers validate types via parsing. This is undesirable for our purposes, however, since it prevents defining types not expressible with a grammar. Instead, we use kinds.

To check kinds, we use the same type checking technique from our term-checking macros. Figure 8 defines a single kind named \( \#\text{type} \) and all types are tagged with this kind (e.g., line 8). Thus, \( \to \) and \( \lambda \) may validate their input types with \( \text{valid-} \tau \) (lines 6-7, 11-12). The use of the macro expander to validate types also differentiates when a type is undefined, rather than malformed. Ultimately, the previous examples now produce type errors:

\[
\begin{align*}
\text{TYERR}: & \quad \text{requires } >=1 \text{ args} \\
\text{TYERR: } & \quad \text{unbound id } \text{Undef}
\end{align*}
\]

**4. A Metalanguage for Typed Languages**

### 4.1 Interleaved Type Checking and Rewriting

Section 3's STLC implementation reveals a synergy between macro expansion and type checking in that Racket’s macro infrastructure can be reused to also check and erase types during its program traversal. Figure 9 refines figure 8 to incorporate this reuse. This organization further suggests a reformulation of figure 8’s rules to combine type checking and erasure, shown in figure 10. A new \( \Gamma \vdash e \to \vec{\tau} \) rule reads “in context \( \Gamma \), \( e \) erases to \( \vec{\tau} \) and has type \( \tau \)”, where contexts consist of variable “erasures”, e.g., \( \text{TE-ABS} \) inserts a binding indirection level in the context in order to add type information for variables and checks a \( \lambda \) body in this context. These rules straightforwardly correspond to our macro-based type system implementation in section 3 where \( \Gamma \vdash e \to \vec{\tau} \) is implemented as “in context \( \Gamma \), \( e \) expands to \( \vec{\tau} \), with type \( \tau \) attached”. Since this paper focuses on implementation, we do not formally study these new typing rules, though they do suggest how to further improve our approach to implementing typed embedded languages.

### 4.2 The TURNSTILE Metalanguage

Section 3 demonstrates that a macro system’s infrastructure can be reused to implement typechecking. Deploying such an approach, however, requires writing macro-level code to embed type rules into macro definitions despite the resemblance of this code to its mathematical specification. This section introduces TURNSTILE, a Racket DSL for creating practical embedded languages that abstracts the macro-level ideas and insights from the previous section into linguistic constructs at the level of types and type systems.
\[
\begin{align*}
\tau ::= & \ldots, e ::= \ldots, \Gamma ::= \mathcal{F} \mid \lambda \tau. \mathcal{F} \mid \mathcal{F} \\
\Gamma, x \gg \tau_1 : \tau_2 \quad &\quad\text{(TE-VAR)} \\
\Gamma, x \gg \tau \gg \tau \quad &\quad\text{(TE-ABS)} \\
\Gamma \gg \lambda \tau_1. \mathcal{F} : \tau_1 \rightarrow \tau_2 \\
\Gamma \gg \mathcal{F} : \tau_1 \rightarrow \tau_2 \\
\end{align*}
\]

Figure 10. Interleaved typechecking and erasure rules

Specifically, TURNSTILE enables writing rules using the syntax from figure 10 but with bidirectional \texttt{\#app} "synthesize" (\(\Rightarrow\)) and "check" (\(\ll\)) arrows replacing the colon, to further clarify inputs and outputs. Figure 11 reimplements STLC with TURNSTILE. TURNSTILE repackages all the infrastructure from section 3 as convenient abstractions, e.g., define-type-constructor \texttt{(d-t-c)} on line 2 and the subsequent define-typerules \texttt{(d-t)} that implement \texttt{\#app} and \(\lambda\).

TURNSTILE’s syntax further demonstrates the connection between specification and implementations enabled by our macro-based approach. Though programmers may now write with a declarative syntax, STLC’s implementation has not changed as TURNSTILE’s abstractions are mere syntactic sugar for the macros from section 3. For example, \(\Rightarrow\) abbreviates \#with used with \texttt{comp+erase-\(\tau\)} and thus figure 11 line 4 exactly corresponds to figure 7 line 4. Similarly, \(\ll\) abbreviates \#fail-unless, \#with, and \(\tau\) so figure 11 line 5 corresponds to figure 7 lines 5-8. Finally, \(\ll\) below the conclusion line corresponds to \texttt{add-\(\tau\)} as in figure 7 line 9 (crossing the conclusion line inverts the yellow and gray positions of \(\gg\)). The \(\lambda\) \texttt{d-t’s} premise computes \(\texttt{e}'\texttt{s} type in a type context containing the variables to the left of \(\gg\) (figure 11 line 9). In addition, the \(\lambda\) input pattern (line 8) utilizes annotations asserting that \(\chi\) is an identifier and \(\tau_{\text{in}}\) is a valid type.

In general, a \texttt{d-t} resembles a figure 10 rule except the conclusion is split into its inputs and outputs—the (yellow) pattern(s) (and \(\gg\)) that begin a definition, and the (gray) syntax following the conclusion line, respectively—such that the definition (and variable scoping) reads top-to-bottom. Figures 10 and 11 additionally differ because \texttt{d-ts} do not explicitly thread through a \(\Gamma\), a consequence of reusing Racket’s scoping for the type environment. Thus TURNSTILE programmers only write new type environment bindings in \texttt{d-ts}, analogous to \texttt{let}; existing bindings are implicitly available according to standard lexical scope behavior.

4.3 Reusing a Type System

TURNSTILE type rules from one language may be reused in the implementation of another. Though the STLC language implements function application and \(\lambda\), it defines no base types and thus no well-typed programs. We next add integers and addition but instead well-typed programs. We next add integers and addition but instead

Viewed as type rules, figure 11 appears to be missing the \(\ll\) rules. While a programmer may write explicit \(\ll\) rules (see 6), in their absence, TURNSTILE uses this default:

\[
\begin{align*}
(\text{define-typerule} & \quad e \ll \tau) \gg \\
&\quad [\Gamma \gg \tau \gg \tau_{\text{in}}] \\
&\quad [\Gamma \gg \tau_{\text{in}} \gg \tau_{\text{out}}] \\
\end{align*}
\]

This implicit definition corresponds to figure 7 lines 5-8. The first and last lines again comprise the input and output components of the rule’s "conclusion", respectively, with the "expected" type now a part of the input pattern matching.

Though TURNSTILE programmers may implement type rules in a declarative style, such a style may be insufficient for creating practical languages, e.g., they do not allow specification of detailed error messages. Therefore, all the macro features from section 3 are also available to a \texttt{d-t} definition, giving TURNSTILE type rules access to the full power of Racket’s macro system. For example, a programmer may add \#fail-unless error messages as in figure 7. Here is a refined \texttt{\#app} that further differentiates arity errors:

\[
\begin{align*}
(\text{define-typerule} & \quad (\text{\#app} \texttt{\_fn} \texttt{\_arg} \ldots) \gg \\
&\quad [\Gamma \gg \texttt{\_fn} \gg \tau] \\
&\quad [\Gamma \gg \tau \gg \tau_{\text{in}} \\
&\quad [\Gamma \gg \tau_{\text{in}} \gg \tau_{\text{out}}] \\
&\quad [\Gamma \gg \texttt{\_arg} \gg \tau] \\
&\quad [\Gamma \gg \tau \gg \tau_{\text{in}} \ldots \tau_{\text{out}}] \\
\end{align*}
\]

4.3 Reusing a Type System

TURNSTILE type rules from one language may be reused in the implementation of another. Though the STLC language implements function application and \(\lambda\), it defines no base types and thus no well-typed programs. We next add integers and addition but instead of revising STLC, we reuse its rules in a new language, analogous to section 3. Specifically, STLC+PRIM in figure 12 uses STLC as a library, importing and re-exporting its type rules with \texttt{extends}. To STLC’s definitions, STLC+PRIM adds an \texttt{Int} base type (line 4), \(\texttt{a+primop}\) (line 5), and integer literals (lines 7-10). Just as the macro expander inserts \texttt{\#app} before applied functions, it also wraps literals with \texttt{\#datum}, whose behavior is overridden in figure 12 to add types to integers. With STLC+PRIM, we can now write well-typed programs.
5. A Series of Core Languages

To confirm that our approach to typed languages handles a variety of type systems, we implemented a series of textbook core languages [8]. This section describes a few examples.

5.1 Types That Bind: Existential Types

Figure 13 depicts EXIST, a language with existential types; it reuses records and variants from another language. The #:bvss option (line 2) specifies that an existential type binds one variable and thus has surface syntax \( \exists \mathcal{X} \mathcal{T} \).

Figure 4 (line 8) introduced type equality as structural equality of syntax objects. Type equality of quantified types, however, must additionally consider alpha equivalence. While other systems commonly convert to alternate representations such as de Bruijn indices [3] to implement this behavior, our use of syntax objects for types remains sufficient since these objects already contain knowledge of the program’s binding structure. Thus the \( \tau = \) used by TURNSTILE looks like:

\[
(\text{define } (\exists \mathcal{X} \mathcal{T} = \tau) (\exists \mathcal{Y} \mathcal{T} = \tau))
\]

This updated \( \tau = \) function specifies multiple input patterns. The first clause matches binding types where equality with the same constructor is equivalent to renaming parameter references to the same name and recursively comparing the resulting body for equality. Otherwise, types are structurally compared. A subst function performs this renaming:

\[
(\text{define } (\text{subst } \mathcal{V} \mathcal{X} \mathcal{E}))
\]

Specifically, subst \( \mathcal{V} \mathcal{X} \mathcal{E} \) replaces occurrences of \( \mathcal{X} \) in \( \mathcal{E} \) with \( \mathcal{V} \), where binds? determines “occurrence” by examining lexical information in the syntax objects. Thus substitution is a structural traversal and no renaming is necessary.

The pack and open macros use \( \tau = \) and subst: pack assigns a term \( \mathcal{E} \) an existential type \( (\exists \mathcal{X} \mathcal{T} = \tau) \), where \( \mathcal{E} \) has concrete type equal to replacing \( \mathcal{X} \) in \( \mathcal{T} = \tau \) with \( \mathcal{E} \). Dually, open binds \( \mathcal{X} \) to an existentially-typed \( \mathcal{E}_{\text{packed}} \)'s value, type variable \( \mathcal{X} \) to \( \mathcal{E}_{\text{packed}} \)'s hidden type, and then checks an expression \( \mathcal{E} \) in the context of \( \mathcal{X} \) and \( \mathcal{X} \). To the left of \( \tau = \) (figure 13, line 9) is two environments: a list of type variables and the standard environment for term variables. The \( (\exists \mathcal{Y} \mathcal{T} = \tau) \) type of \( \mathcal{E}_{\text{packed}} \) is “opened”, so \( \mathcal{X} \) has type \( \mathcal{T} \) but with occurrences of the existentially-bound \( \mathcal{Y} \) (not in scope in \( \mathcal{E} \)) replaced with its “opened” \( \mathcal{Y} \) name. Here is a typical counter example (\( \tau = \text{RCRD} \), and \( \mathcal{E} \) correspond to records):

5.2 Subtyping and Enhanced Modularity

Figure 14 presents STLC+SUB, a language with subtyping that reuses parts of STLC+PRIM from figure 12 but adds new base types and redefines \#%datum and + with these types. One might not expect STLC+SUB to be able to reuse type rules that do not consider subtyping. However, TURNSTILE exposes hooks for common type operations and implements type checking in terms of these hooks, enabling better reuse. For example, \( \tau = \) in figure 5, line 4 is actually an overridable “type check relation” (initially set to \( \tau = \)). These language-level hooks are implemented with Racket parameters [19], which allow a controlled form of dynamic binding. Thus STLC+SUB defines a new \( \tau = \) predicate and installs it as the \( \exists \mathcal{X} \mathcal{T} = \tau \) type check relation (we oval-box parameter names), enabling reuse of \#%app and \( \lambda \) from STLC.

TURNSTILE pre-defines parameters like \( \exists \mathcal{X} \mathcal{T} = \tau \) and \( \tau = \) eval\}: the latter is called before attaching types to syntax. Each language may also define new parameters, e.g., STLC+SUB additionally defines \#%app and uses it in conditionals.

5.3 Defining Types and Kinds

We implemented a term to type check and erase its types. We can check kinds the same way: expanding a type kind checks and erases kinds. The kind erasing may cause problems, however, since a type judgement may use both types and kinds. Nevertheless, TURNSTILE can define a kind system like in FC. To address the problem, figure 15 reformulates some FC rules with our \( \gg \) relation. Specifically, T-TABS and K-ALL erase a V’s kind annotation, but “save” it with \( \gg \), now a kind constructor, in the same manner that \( \gg \) “saves” a \( \lambda \)’s type annotations. T-TAPP then checks that its argument type has a kind matching the saved annotation.

Figure 16 implements FOMEGA utilizing figure 15’s insights: it introduces a new “kind” category of syntax, defines \( \gg \) and \(* \)
| feature / lang name | λ | app | → | τ= | τeval | Int | + | datum | Bool if | letrec | begin | define | alias | tup | type | tup | recrd | variant | List | Ref | Top | Nat | τ<: | join | μ | ∃ | ∀ | Λ | inst | tyapp | tyλ | invoke | kind |
|---------------------|---|-----|---|----|-------|-----|---|-------|-------|-------|-------|-------|-----|----|----|----|------|-------|-----|----|----|----|-----|----|----|----|----|------|--------|-----|----|
| stlc                | ⧫ | ⧫  | ⧫ | ⧫  | ⧫    |     |   |       |       |       |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |     |
| stlc + prim         |   |     |   |     |       |     |   |       |       |       |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |
| extended stlc       |   |     |   |     |       |     |   |       |       |       |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |
| tuples              |   |     |   |     |       |     |   |       | ⧫    | ⧫    |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |     |
| records + variants  |   |     |   |     |       |     |   |       |       |       |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |
| lists               |   |     |   |     |       |     |   |       |       |       |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |
| reference cells     |   |     |   |     |       |     |   |       |       | ⧫    |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |
| subtyping           |   |     |   |     |       |     |   |       | ⧫    | ⧫    |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |
| subtyping + records |   |     |   |     |       |     |   |       | ⧫    | ⧫    |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |
| (iso) recursive     | ⧫ | ⧫  |   |     |       |     |   |       |       |       |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |
| existential         |   |     |   |     |       |     |   |       | ⧫    | ⧫    |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |
| system F            |   |     |   |     |       |     |   |       | ⧫    | ⧫    |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |
| F<:                 | ⧫ | ⧫  |   |     |       |     |   |       |       |       |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |
| F omega             | ⧫ | ⧫  |   |     |       |     |   |       |       |       |       |       |     |     |    |     |       |       |     |     |     |     |     |     |     |     |     |     |

○ = implemented here  ⊕ = extends the above
A unique color represents each language. The features in each language (row) are colored according to the language where they are defined.

Table 1. Implemented Languages
and \texttt{ref}, \texttt{deref}, and \texttt{:=} type rules for allocation of, dereference of, and assignment to reference cells, respectively (\texttt{box} is Racket’s \texttt{ref} cells). In addition to types, the language tracks source locations (\texttt{loc}) of \texttt{ref} allocations (line 9). The \texttt{ref} rule exhibits new syntax: instead of a type to the right of \texttt{:=}, a programmer may write multiple \texttt{⇒} arrows matching multiple properties. Thus \texttt{ref} specifies that expansion of \texttt{e} (line 6) computes both a type (keyed on \texttt{e}) and a set of locations \texttt{π} (keyed on \texttt{e}). The key symbols match the user-specified symbols below the conclusion line. The \texttt{⇒} rule uses both \texttt{⇒} (for the type) and \texttt{⇒} (for the locations) simultaneously (line 16).

\textbf{Effect} contrasts with Table 1’s languages in that it cannot reuse \texttt{#%app} and \texttt{λ} due to its incompatible type relation. (It does reuse some types and type operations.) The new \texttt{#%app} and \texttt{λ} rules show that both terms and types carry the \texttt{∶} property. Specifically, \texttt{λ} propagates \texttt{∶} to function types (line 30), expressed with a \texttt{nested} \texttt{⇒} (like the double-\texttt{⇒} syntax for kinds from Figure 16), because evaluating a \texttt{λ} does not trigger allocations in its body. Applying a function does evaluate the body, so \texttt{#%app} transfers locations from the function type (line 21) to the application term (line 26).

6. A Full-Sized Language

To show that \textsc{turnstile} scales to real-world type systems, we created \textsc{mlish}, an ML-like language with local type inference, recursive user-defined algebraic data types, pattern matching, and basic Haskell-style type classes [17], along with “batteries” such as efficient data structures, mutable state, generic sequence comprehensions, I/O, and concurrency primitives. \textsc{mlish} also demonstrates how \textsc{turnstile} easily incorporates type-system-directed program transformations. This section explains a few features.

\textbf{Local type inference} \textsc{mlish} aims to follow Pierce and Turner’s empirical inference guidelines [19]. Specifically, programmers need not write most annotations and instantiations except top-level function signatures, which are useful as documentation, and some \texttt{λ} annotations, which are rare.

Figure 18 sketches basic type inference in \texttt{λ} and \texttt{#%app}. Multiple clauses comprise \texttt{λ}, whose input patterns are checked in order. The first clause matches unannotated \texttt{λ}s whose context determines its type, indicated with \texttt{⇒} (line 4). The second matches annotated \texttt{λ}s with implicitly bound type variables, computes these variables, and then recursively invokes the \texttt{λ} rule (indicated with \texttt{⇒}) with explicit type variables. In this manner, a surface language with implicit type variables rewrites to one with explicit binders, reusing the macro system for the type-system-directed rewrite. Finally, the third clause matches \texttt{λ}s with explicit type variable binders; it resembles \texttt{λ} from Figure 11. An \textsc{mlish} \texttt{define} for top-level functions uses \texttt{λ}, splitting a definition into a runtime component and a macro that adds type information:
To implement a \( \Leftarrow \) type rule, e.g., figure 18 lines 4-7, MLISH propagates "expected type" information from an expression's context by attaching a syntax property before expansion, making the information available while type checking that expression. A \( \Leftarrow \) type rule's input matches this expected type (line 4), and also implicitly attaches it to the output syntax (line 7). A non-\( \Leftarrow \) type rule may also inspect the expected type, as with \#%(app clause. Specifically, the first \#%(app clause extracts the expected type (line 22) and uses it to solve for the type variables (line 26). The clause then recursively invokes \#%(app with explicit instantiation types. In this manner, a surface language with inferred instantiation rewrites to one with explicit instantiation. The second \#%(app clause resembles the first except it does not use the expected type. The third instantiates the polymorphic type function (line 39) and then checks the function arguments as in figure 11.

**Algebraic datatypes** Figure 19's \( \text{define-type} \) macro defines sum-of-product datatypes in MLISH; it expands to a series of definitions (gray box): a type constructor (lines 3-5), where the \#extra argument communicates information about the type to other type rules, e.g., to check match clause completeness; Racket structs (line 6) implementing runtime constructors; and type rules (lines 7-11) that leverage \#%(app to instantiate polymorphic constructors.

**Pattern matching** In figure 19's \( \text{match} \), one of more clauses follows \( e_\text{n} \), matching its possible variants. The rule uses "extra" information from the type to check clause exhaustiveness (lines 14-16). Otherwise \( \text{match} \) expands to a conditional that extracts components of \( e_\text{n} \) with accessors (also from the "extra" information). Here is an MLISH example:

```
(define-type (Ty X ...)
  (Constr [fld : (\tau X)] ...))

(define-type-constructor Ty
  #:arity (len (X ...))
  #:extra ((Constr [fld : (\tau X)] ...))
  (struct Constr intra (fld ...))
  (define-type-rule (Constr e arg ...) =>
    (#with C (add-\tau Constr intra)
      (\tau X ...) \rightarrow (\tau ...)
      (Ty X ...))))

---

(define-typerule (match e with [C (x ... -> e ...)])
  [\i -> e \Rightarrow \tau \Rightarrow \tau X]
  #:with [(\tau X) C (x ...)]
  (cond [(\tau X) ?]
        (let ([\tau (get-fld ?) ...]) ...))
  => (first \tau ...))

---

```

**Type classes** Figure 20 sketches an implementation of type classes. The rules interleave typechecking and program rewriting, demonstrating how TURNSTILE naturally accommodates such interleaving. MLISH type classes only support basic features such as subclassing (unsupported features include multi-parameter type classes and overlapping instances). For simplicity, this paper shows an implementation is not shown) have a typeclass component in their usage. MLISH supports the general multi-operation version.

The \( \text{define-tc} \) form shows how two definitions implement a class: a macro for the type class itself (line 1) that expands to its generic operation and type, and a type rule for that operation (lines 3-7) that looks up a concrete operation (line 5) based on the generic name and the concrete types of its usage. MLISH type classes reuse the compile-time macro environment for lookups, where a concrete operation’s name, installed by \( \text{define-instance} \) (lines 8-14), is a mangling of the generic name and specific concrete types.

Consequently, functions utilizing generic operations (this \lambda implementation is not shown) have a typeclass component in their type (the \Rightarrow constructor on line 18) and these functions implicitly
have an extra concrete operation argument. The `#%app` rule implicitly inserts this argument by: extracting the generic operation of the type class (line 21); looking up the concrete operation based on instantiation types for the function (lines 22-23); and adding this operation to the application (line 25).

### 7. Creating a Test Suite

Sections 5 and 6 show that our approach accommodates a variety of typed languages. This section explains how we validate these languages with a test suite of real-world programs [1, 31, 34]. Our tests utilize TURNSTILE’s unit-testing framework, which accommodates testing of typechecking successes, failures, as well as error messages. The testing framework also allows all tests to be written with a language’s surface syntax, rather than an internal AST structure. The following example defines a function `f`, tests the type of `f`, and both a successful and failing application of `f`:

```scheme
#lang mlish (require typechecker-tester)
(define (f [x : Int] -> Int Int)
  (check-type f (~> Int Int))
  (check-type (f 1) : (Int Int))
  (check-type-fail (f 1 2) #:msg "Wrong number of args")
```

Table 3 summarizes our test suite, which includes both “coverage” tests checking general functionality and corner cases, and real-world examples. For the latter, Real World OCaml [31] supplied functional tests while the Benchmarks Game [1] consisted of more imperative tests. Okasaki’s data structures tested the limits of our type system. For example, in discussing polymorphic recursion (chapter 10), Okasaki writes:

> “We will often present code as if SML supports [polymorphic recursion]. This code will not be executable but will be easier to read.”

We were able to add polymorphic recursion to MLISH, by leveraging recursive definition forms in the host, and implemented the data structures in question, demonstrating both the ability to implement tricky type system features with TURNSTILE, and the ease with which one can do so.

### 8. Related Work

Many researchers have developed extensible type systems [2][4][11][28][30][36][37]. These frameworks typically augment a fixed host type system, which imposes some limitations on what kinds of extensions are allowed. For example, some do not allow defining new types, while others may only define new rules expressible with the host system. Though ease of extension is a feature of TURNSTILE-created languages (any language in the Racket ecosystem may serve as TURNSTILE’s host language, so TURNSTILE may also be used to extend a typed language like the other systems), it is not the sole focus of our work. Instead, we wish to support the creation of complete languages that may utilize arbitrary type rules.

Others have devised special-purpose macro systems for building type checkers [13][35]. Whether these systems accommodate embedded DSL creation, however, remains undetermined. Our work takes the opposite approach. We start with a popular platform for creating embedded languages and show that its general macro system already accommodates type checking.

Typed Racket [42] pioneered the idea of creating a typed language using syntax extensions. While languages created with TURNSTILE share this high-level description, our approach differs in its goals and implementation details. Typed Racket aims to type check Racket programs and thus first expands a program, and then feeds this expanded program to a conventional monolithic type checker that recognizes only core Racket forms (for this reason Typed Racket is considered a “sister” language to Racket rather than an embedded language). In contrast, we wish to support the creation of arbitrary typed surface languages, and we do so via implementations that interleave macro expansion and type checking. This requires programmers to implement type rules for all surface constructs rather than just core forms, however, but TURNSTILE helps this process by providing a concise declarative syntax for writing these rules. Interleaving macro expansion and type checking yields the additional benefit of using type information during expansion, allowing types to direct a macro’s output.

The TinkerType [29] system also separates type rules and operations into reusable components. The framework combines raw strings rather than linguistic components, however, and is designed for modeling and typesetting calculi rather than creating practical languages. Nonetheless, our approach may benefit from some of TinkerType’s consistency checks when combining components.

### 9. Conclusions and Future Work

We present a novel use of macros to create practical typed embedded languages. Our approach is not constrained to a particular type system, yet programmers do not have to implement a system from scratch because they can reuse the infrastructure of a macro system. To this end we introduce TURNSTILE, a metalanguage for creating typed languages using a declarative type-and-rewriting rule syntax. We conjecture that language implementers will benefit from our approach, as non-experts may reduce the burden of language creation, while researchers may rapidly iterate and experiment with new type features and combinations of features.

We next plan to further validate our idea by implementing more languages, and to extend our approach to more complex analyses. In addition, we plan to explore whether our approach to implementing typed embedded languages is compatible with other, non-Lisp-style syntax extension systems. Finally, we plan to investigate whether the connections between type checking and macro processing that we have described might inform the future design of both kinds of systems. The fact that many type systems already interwove type checking and program rewrites [19][20][47] suggests that perhaps languages should come equipped with a general framework for defining macros and type rules, as well as some combination of the two.

### Acknowledgments

This paper is supported by NSF grant SHF 1518844. We thank Asumu Takikawa, Matthias Felleisen, and our reviewers for feedback on drafts, Ryan Culpepper for technical discussions about macros, and Alexis King for suggestions on language design.

### References

1. The computer language benchmarks game. URL http://benchmarksgame.alioth.debian.org/