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 performant 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...” [23]. 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 resembles 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, 15], 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 [14] and has even influenced macro system design in modern non-Lisp languages [6, 8, 10, 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 (§2-3) 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 (§4) introduces TURNSTILE, a metalanguage that abstracts the insights and techniques from the first part into convenient linguistic constructs. The third part (§5-7) 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:

```racket
#lang racket
(define-m (lm x e) (\(x\) e))
(provide lm)
```

1 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/
A `#lang racket` declaration allows LAM to use forms and functions from the main Racket language. LAM defines and exports one macro,\(^2\) `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 subpieces of the input, the parameter and body, respectively. The output syntax `(λ (x) 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
(define-m (checked-λ ...) (when T-Abs) er(_:))
(define-m (checked-app ... ) (when T-App) er(_:))
```

![Figure 2. Simply-typed λ-calculus](image)

![Figure 3. A type checking function application macro](image)

\[^2\]define-m abridges Racket’s `define-syntax` and `syntax-parse`[9].

A `#lang lc` declaration allows LC to use forms and functions from the main LC language. LC defines and exports one macro, `lc`, denoting single-argument functions. A Racket macro consumes and produces syntax object data structures. The `lc` macro specifies its usage shape with input pattern `(lc x (lm y x))` (in yellow to help readability), which binds pattern variables `x` and `e` to subpieces of the input, the parameter and body, respectively. The output syntax `(λ (x x))` (gray denotes syntax object construction) references these pattern variables (`λ` is Racket’s `λ`).

A module declaring `#lang lc` 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 lc)
(provide (rename-out [lm λ] [app #%app]))
(define-m (app e_in e_arg) (λ arg (%app e_in 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 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 type checks the syntax tree (“v0” marks this initial version). Additional `#:with` and `#:when` condition guards 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_in`, 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 `e_arg` satisfy predicate `τ=`. The types in `τ_in` and `τ_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.
(define (add-τ e τ) (add-stx-prop e 'type τ))
(define (get-τ e) (get-stx-prop e 'type))
(define (compute-τ e) (get-τ (local-expand e)))
(define (erase-τ e) (local-expand e))
(define (comp+erase-τ e) ; get e's type, erase types
#:with e (local-expand e) #:with τ (get-τ e)
[Ω τ])
(define (add-τ (τ≡ numnum τ) (stx= τ1 τ2))

Figure 4. Helper functions for type checking

(define-m (checked-app efn earg) ; v1
#:with [efn (→ τin τout)] (comp+erase-τ efn)
#:with [earg τarg] (comp+erase-τ earg)
#:when (τ = τarg τin)
(add-τ (λ%app efn earg) τout))

Figure 5. Revise fig 3 to compute and erase types together

3.2 Communicating Macros

The organization of checked-app in figure 3 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 1) 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 [16] 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 figure 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:

(require (rename [→ a])) (τ≡ (a ≡ t) (→ s t)); => 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 → types (lines 1-2) as a macro that matches on an input and output type and expands to an application of an internal function that returns an error at runtime (there are no base types for now, see §4.3). 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-macro1 (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 unbound y 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 x), the type-erased π, 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

1let-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 \(\tau_{in}\) and ellipses follow \(\tau_{in}\) in its output. The other forms are extended similarly. The \(\text{checked-}\lambda\) macro uses a slightly modified \(\text{comp+erase-}\tau/\text{ctx}\) (line 13) that accepts multi-element contexts. In \text{checked-app} (line 5), the “vector” \(\vec{\tau}\) notation denotes \(\tau\) mapped over its input list.

**Error messages** Figure 7 also reports more useful error messages. The \text{checked-app} in figure 5 reports type errors as syntax errors but a better message should indicate the error’s location and the computed expected types. The \text{checked-app} in figure 7 reports type errors as syntax errors but a better message should indicate the error’s location and the computed expected types. The \text{checked-app} (lines 6-8) produces 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., \(\lambda (\text{[x : (\to\to)]})\ x\) or \(\lambda (\text{[x : Undef]}\ x)\) 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:

\[
(\lambda (\text{[x : (\to\to)]})\ x)\ \text{TYERR: requires } \geq 1 \text{ args}
\]
\[
(\lambda (\text{[x : Undef]}\ x)\ )\ \text{TYERR: unbound id Undef}
\]

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 1 to incorporate this reuse. This organization further suggests a reformulation of figure 2’s rules to combine typechecking and erasure, shown in figure 10. A new \(\Gamma \vdash e : \tau\) rule reads “in context \(\Gamma\), \(e\) erases to \(\tau\) and has type \(\tau\)”, where contexts consist of variable “erasures”, e.g., 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 \(\Gamma \vdash e : \tau\) is implemented as “in context \(\Gamma\), \(e\) expands to \(\tau\), 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 Metametalanguage**

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.
\[ \tau ::= \ldots \quad e ::= \ldots \quad \mathbf{F} ::= \mathbf{F} \mid \lambda \mathbf{x}. \mathbf{F} \mid \mathbf{F} \mathbf{F} \quad \Gamma ::= \mathbf{F} : \tau, \ldots \]

\[
\begin{align*}
\text{(TE-VAR)} & \quad \Gamma, x \gg \tau_1 \vdash e \gg \tau_2 & \quad \pi \notin \Gamma & \quad \Gamma \vdash e \gg \tau_1 : \tau_2 \\
\text{(TE-ABS)} & \quad \Gamma, x \gg \tau : \tau_1, e \gg \lambda \mathbf{x}. \mathbf{F} : \tau_1 \rightarrow \tau_2 & \quad \pi \notin \Gamma & \quad \Gamma \vdash e \gg \tau_1 : \tau_2 \\
\text{(TE-APP)} & \quad \Gamma, e_1 \gg \tau_1 : \tau_1 \rightarrow \tau_2, \Gamma, e_2 \gg \tau_2 : \tau_2 & \quad \Gamma \vdash e_1 \mathbf{F} e_2 \gg \tau_1 \mathbf{F} \tau_2 \\
\end{align*}
\]

\[\tau \text{ extends } \mathbf{F}\]

\[\text{(define-base-type Int)}\]

\[\text{(define-primop + : } (\rightarrow \text{ Int Int Int}))\]

\[\text{(define-typerule } (#\text{datum n}) \Rightarrow \text{Int})\]

\[\text{Figures 10 and 11 renumeral the type environment. Thus programmers may now write explicit } \Rightarrow \text{ rules. While a programmer may write explicit } \Rightarrow \text{ rules (see §6), in their absence, TURNSTILE uses this default:}\]

\[\begin{align*}
\text{(define-typerule } e \Rightarrow \tau & \Rightarrow \tau_2) \\
\text{[} & \quad e \gg \tau \Rightarrow \tau_2 \\
\text{]} & \quad \tau_2 = \tau \\
\text{[} & \quad \tau \gg \tau_2 \\
\end{align*}\]

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.

TURNSTILE’s syntax further demonstrates the connection between specification and implementation 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 \(\,:\,\text{with}\) used with \texttt{comp-erase-\tau}\) and thus figure 11, line 4 exactly corresponds to figure 7, line 4. Similarly, \(\Leftrightarrow\) abbreviates \#:fail-unless, \#:with, and \#\tau so figure 11, line 5 corresponds to figure 7, lines 5-8. Finally, \(\Rightarrow\) below the conclusion line corresponds to add-\tau as in figure 7, line 9 (crossing the conclusion line inverts the yellow and gray positions of \(\Rightarrow\)).

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 re-use its rules in a new language, analogous to section 2. 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 [38]. 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 #: bvs option (line 2) specifies that an existential type binds one variable and thus has surface syntax (\( \exists X \) \( \text{body} \)).

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 [5] 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:

\[
(\text{define } ((\exists X \text{ body})) X)
\]

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 sub expression performs this renaming:

\[
(\text{define } \text{subst} \text{ v x e})
\]

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

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

\[
(\text{define-primop + : (→ Nat Nat) → Nat })
\]

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 \( \tau \)).

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 \( \tau \); precinct and installs it as the \( \tau \) type check relation (we ovbox-param parameter names), enabling reuse of \#app and \lambda from STLC.

\[
(\text{define COUNTER}
\]

Specifically, \( \text{pack} \text{ v x e} \) replaces occurrences of \( x \) in \( e \) with \( v \), where \( \text{binds} \) determines “occurrence” by examining lexical information in the syntax objects. Thus substitution is a structural traversal and no renaming is necessary.

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

\[
(\text{define COUNTER}
\]

5.3 Defining Types and Kinds

We implement macro-expand a term to type check and erase its types. We can check kinds the same way: expanding a type kind produces an overridable “type check relation” (initially set to \( \tau \)).

The latter is called before attaching types to syntax. Each language may subsume parts of STLC, K-ALL erase a \( \lambda \)'s kind annotation, but “save” it with \( \ast \), now a kind constructor, in the same manner that \( \rightarrow \) “saves” a \( \lambda \)'s type annotations. T-TAPP then checks that its argument type has a kind matching the saved annotation.

\[
(\text{define-kind} \text{app} \text{type})
\]

Figure 16 implements FOMEGA utilizing figure 15’s insights: it introduces a new “kind” category of syntax, defines \( \Rightarrow \) and \( \ast \)
Table 1 summarizes extensions and reuse in fourteen core language implementations. A row and color represents a language and features are in columns. A diamond marks a feature’s first implementation and down-column appearances of the feature’s color indicates reuse. Thus single-color columns and multi-color rows indicate abundant reuse. For example, all languages share the same \( \lambda \); also, languages with basic types share a \( \tau \) while those with binding types use an extended version. A \( \oplus \) marks feature extension types. Finally, the \( \Delta \) rule type checks its body in its type variable’s context, and the \( \xi \) rule instantiates an expression \( e \) at type \( \tau \) by computing \( e \)’s \( \forall \) type and that type’s kind \( (\forall \kappa) \), and checking that \( \tau \) has kind \( \kappa \).

### 5.4 Reusing Languages

Table 1 summarizes extensions and reuse in fourteen core language implementations. A row and color represents a language and features are in columns. A diamond marks a feature’s first implementation and down-column appearances of the feature’s color indicates reuse. Thus single-color columns and multi-color rows indicate abundant reuse. For example, all languages share the same \( \lambda \); also, languages with basic types share a \( \tau \) while those with binding types use an extended version. A \( \oplus \) marks feature extension types. Finally, the \( \Delta \) rule type checks its body in its type variable’s context, and the \( \xi \) rule instantiates an expression \( e \) at type \( \tau \) by computing \( e \)’s \( \forall \) type and that type’s kind \( (\forall \kappa) \), and checking that \( \tau \) has kind \( \kappa \).

![Figure 15. Some \( F_\omega \) rules using figure 10’s \( \Leftrightarrow \) relation](image)

![Figure 16. A language with higher-order polymorphism](image)

(a dotted line connects non-adjacent-row extensions). For example, “subtyping” extends \( \#%\text{datum} \) from “stlc+prim”; also, \( F_\omega \) extends and combines subtyping from one language and \( \forall \) from system \( F \) to implement bounded polymorphism.

Table 2 summarizes implementation sizes from table 1. Each column represents a different implementation of the same language: the first uses \( \text{TURNSTILE} \); the second uses \( \text{TURNSTILE} \) but does not import other implementations; and the third uses plain Racket. Though the last two columns are estimates (2 significant figures)—we did not implement every permutation of every language—they still indicate the degree of reuse. Roughly, the second and first column difference represents the degree to which type rules are reused across many languages, analogous to single-color columns in table 1. Such reuse would be difficult to achieve with conventional type checker implementations. The third and first column difference indicates the degree to which \( \text{TURNSTILE} \) captures common patterns used to implement type checkers.

### 5.5 More Than Types: A Type-and-Effect System

Table 1’s languages mostly use a typical \( \Gamma \vdash e : \tau \) relation though \( \text{TURNSTILE} \) is not limited to this relation. Rather, programmers may specify propagation of any number of arbitrary properties. For example, figure 17 presents \( \text{EFFECT} \), a language with a basic type and effect system [33]. The language adds \{Void and Ref\} types,
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 ref. deref, and § := type rules for allocation of, dereference of, and assignment to reference cells, respectively (box is Racket’s ref cells). In addition to types, the language tracks source locations (τ) of ref allocations (line 9). The ref rule exhibits new syntax: instead of a type to the right of :=, a programmer may write multiple ⇒ arrows matching multiple properties. Thus ref specifies that expansion of e (line 6) computes both a type (keyed on ::) and a set of locations τ (keyed on ::.). The key symbols match the user-specified symbols below the conclusion line. The := rule uses both ⇐ (for the type) and ⇒ (for the locations) simultaneously (line 16).

Effect contrasts with table 1’s languages in that it cannot reuse stlc+app and λ due to its incompatible type relation. (It does reuse some types and type operations.) The new stlc+app and λ rules show that both terms and types carry the ⇐, property. Specifically, λ propagates ⇐, to function types (line 30), expressed with a nested ⇒ (like the double⇒ syntax from figure 16), because evaluating a λ does not trigger allocations in its body. Applying a function does evaluate the body, so stlc+app transfers locations from the function type (line 21) to the application term (line 26).

6. A Full-Sized Language
To show that TURNSTILE scales to real-world type systems, we created MLISH, an ML-like language with local type inference, recursive user-defined algebraic data types, pattern matching, and basic Haskell-style type classes [47], along with “batteries” such as efficient data structures, mutable state, generic sequence comprehensions, I/O, and concurrency primitives. MLISH also demonstrates how TURNSTILE easily incorporates type-system-directed program transformations. This section explains a few features.

Local type inference MLISH aims to follow Pierce and Turner’s empirical inference guidelines [39]. Specifically, programmers need not write most annotations and instantiations except top-level function signatures, which are useful as documentation, and some λ annotations, which are rare.

Figure 18 sketches basic type inference in λ and stlc+app. Multiple clauses comprise λ, whose input patterns are checked in order. The first clause matches unannotated λ whose context determines its type, indicated with ⇐ (line 4). The second matches annotated λs with implicitly bound type variables, computes these variables, and then recursively invokes the λ rule (indicated with ⇒) 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 λs with explicit type variable binders; it resembles λ from figure 11. An MLISH define for top-level functions uses λ, splitting a definition into a runtime component and a macro that adds type information:

```
(define-typerule λ ; no annotations, used expected type
  [(λ (x id ... ) e) ⇐ (λ (x id ... ) (→ τ in ... τ out ) ) ]⇒
  ([x ... ] ( [x ∋ x : τ in ... ] ... ) ) ⇒ e ⇒ τ out ] )

; variable annotations, with free tyvars
  [(λ (x id : τ in ... ) e ) ] ⇒
  ; variable annotations, explicit tyvar binders
  [(λ (x ... ) ( [x id : τ in ... ] ... ) ) ⇒
  ...
  ...
  ...
  ...
  ![Figure 18. Type inference in MLISH stlc+app and λ](image)
```

```
(define-typerule (define (f [x : τ ] ... ) → τ out ) e) \implies
  #:with τ in τ out
  (free-tyvars (τ ... ) )

[!- (λ (x ... ) e) ⇒ τ in τ out ] )⇒

; define f intra

(define f intra (x y ... ) ) ⇒
```
To implement a $\Leftrightarrow$ 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 $\Leftrightarrow$ type rule's input matches on this expected type, as with $\#\%\text{app}$ syntax. Specifically, the first $\#\%\text{app}$ clause extracts the expected type (line 22) and uses it to solve for the type variables (line 26). The clause then recursively invokes $\#\%\text{app}$ with explicit instantiation types. In this manner, a surface language with inferred instantiation rewrites to one with explicit instantiation. The second $\#\%\text{app}$ clause resembles the first except it does not use the expected type. The third instantiates the polymorphic function type (line 39) and then checks the function arguments as in figure 11.

Algebraic datatypes Figure 19's 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 $\#\%\text{app}$ to instantiate polymorphic constructors.

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

```
(define-m (define-type (Ty X ...))
  (Constr [fld : $\tau_1$] ...))
(define-type-constructor Ty
  #:arity = (len (X ...))
  #:extra ([(Constr [fld $\tau_1$] ...)]))
(struct Constr_intrnl (fld ...) ...)
(define-typerule (Constr $\mathit{e}_\arg$ ...)
  #:with C (add-$\tau$ Constr_intrnl
               (V (X ...)) ($\tau_1$ ... (Ty X ...))))
  ...)
(def-typerule (match $\mathit{e}_n$ with [C X ... -> $\mathit{e}$] ...))
  ...)
  #:fail-unless (set= (fmt (C ...)) (Excep ...))
  (fmt "missing `a") (set-diff (C ...)) (Excep ...)))
  (x : $\tau_{\text{fld}}$) ... | $\mathit{e}$ | $\Rightarrow$ $\tau$ ...)
  #:when (same-$\tau$ ($\tau$ ...))
  ...)
  (let ((V $\mathit{e}$))
    (cond [(C expect? V)
       (let ([($\mathit{e}$ (get-fld V) ...)] $\mathit{e}$)] ...))
  => (first ($\tau$ ...)))
```

Figure 19. Defining types and pattern matching in MLISH

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 single-operation type classes, though MLISH supports the general multi-operation version. The define-tc form shows that two definitions implement a type class: a macro for the type class itself (line 2) that expands to its generic operation and type, and a type rule for that operation (lines 5-7) 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 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

```
(lang mlish
  (define-type (Tree X)
    (leaf [val : X])
    (node [l : (Tree X)] [r : (Tree X)]))
  (define (sum-tr [t : (Tree Int)]) [t] -> Int)
  (match t with
    [node l r -> (+ (sum-tr l) (sum-tr r))]
  ))
  ; TYERR: match: not enough clauses, missing leaf
```

Type classes
work takes the opposite approach. We start with a popular plat-
embedded DSL creation, however, remains undetermined. Our
programming type checkers creation of complete languages that may utilize arbitrary type rules.

It is not the sole focus of our work. Instead, we wish to support the
also be used to extend a typed language [21] like the other systems),
tem may serve as T
ible with the host system. Though ease of extension is a feature of
host type system, which imposes some limitations on what kinds
data structures in question, demonstrating both the ability to im-
We were able to add polymorphic recursion to
MLISH (typecheck-fail (f 1 2) #:msg!

Table 3. Testing TURNSTILE-created languages

<table>
<thead>
<tr>
<th>test description</th>
<th>core langs (§ 5)</th>
<th>MLISH (§ 6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>coverage</td>
<td>4313</td>
<td>2467</td>
</tr>
<tr>
<td>RW OCaml [31]</td>
<td>610</td>
<td></td>
</tr>
<tr>
<td>Benchmarks Game</td>
<td>852</td>
<td></td>
</tr>
<tr>
<td>Okasaki [34]</td>
<td>2014</td>
<td></td>
</tr>
<tr>
<td>Other examples (e.g., nqueens)</td>
<td>559</td>
<td></td>
</tr>
<tr>
<td>total (LOC, incl. comments)</td>
<td>4313</td>
<td>6502</td>
</tr>
</tbody>
</table>

have an extra concrete operation argument. The %app rule implic-
tely 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 accom-
modates 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 struc-
ture. 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) x)
(check-type f : (-> Int Int))
(check-type (f 1) : Int => 1)
(typecheck-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 [polymor. recur-
sion]. This code will not be executable but will be easier to read.”

We were able to add polymorphic recursion to MLISH, by lever-
aging recursive definition forms in the host, and implemented the
data structures in question, demonstrating both the ability to im-
plement 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 defin-
ing new types, while others may only define new rules express-
ible with the host system. Though ease of extension is a feature of
TURNSTILE-created languages (any language in the Racket ecosys-
tem may serve as TURNSTILE’s host language, so TURNSTILE may
also be used to extend a typed language [21] 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 build-
ting 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 plat-
form 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 dif-
fers in its goals and implementation details. Typed Racket aims
type to type check Racket programs and thus first expands a program,
and then feeds this expanded program to a conventional mono-
lithic type checker that recognizes only core Racket forms (for
this reason Typed Racket is considered a “sister” language [43] 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 informa-
tion during expansion, allowing types to direct a macro’s output.
Finally, since our language implementations are just a series of
macros, they are naturally modular and thus easily extended and
reused.

The TinkerType [29] system also separates type rules and op-
erations 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 embed-
ded 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 analy-
ses. In addition, we plan to explore whether our approach to im-
plementing typed embedded languages is compatible with other,
non-Lisp-style syntax extension systems. Finally, we plan to inves-
tigate 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
intervene type checking and program rewrites [39, 40, 47] sug-
ests that perhaps languages should come equipped with a gen-
eral 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 feed-
back on drafts, Ryan Culpepper for technical discussions about
macros, and Alexis King for suggestions on language design.

References

benchmarksgame.alioth.debian.org/.
for implementing pluggable type systems. In Proceedings of the 21st

E. Burmako. Scala macros: Let our powers combine!: On how rich


