Dependent Type Systems as Macros

STEPHEN CHANG, Northeastern University, USA and PLT Group, USA
MICHAEL BALLANTYNE, Northeastern University, USA and PLT Group, USA
MILO TURNER, Northeastern University, USA
WILLIAM J. BOWMAN, University of British Columbia, Canada

We present TURNSTILE+, a high-level, macros-based metaDSL for building dependently typed languages. With it, programmers may rapidly prototype and iterate on the design of new dependently typed features and extensions. Or they may create entirely new DSLs whose dependent type "power" is tailored to a specific domain. Our framework's support of language-oriented programming also makes it suitable for experimenting with systems of interacting components, e.g., a proof assistant and its companion DSLs. This paper explains the implementation details of TURNSTILE+, as well as how it may be used to create a wide-variety of dependently typed languages, from a lightweight one with indexed types, to a full spectrum proof assistant, complete with a tactic system and extensions for features like sized types and SMT interaction.

3

CCS Concepts: • **Software and its engineering** → *Specialized application languages.*

Additional Key Words and Phrases: macros, type systems, dependent types, proof assistants

ACM Reference Format:
Stephen Chang, Michael Ballantyne, Milo Turner, and William J. Bowman. 2020. Dependent Type Systems as Macros. *Proc. ACM Program. Lang.* 4, POPL, Article 3 (January 2020), 29 pages. https://doi.org/10.1145/3371071

1 THE TROUBLE WITH IMPLEMENTING DEPENDENT TYPES

Dependent types are breaking into the mainstream. For example, Scala supports path-dependent types [Amin et al. 2014; Cremet et al. 2006], Haskell has embraced type-level computation [Weirich et al. 2017], and Rust has considered Π types [2017]. Further, interactive languages like F* are increasingly used to verify critical software such as Firefox's TLS [Zinzindohoué et al. 2017].

Unfortunately, widespread use remains on hold as language designers continue exploring the design space, trying to balance the power of dependent types with their steep learning curve. Worse, because dependent types blur the distinction between types and runtime values—complicating a language's implementation as well—evaluating a feature is more difficult and iterating on its design is extremely time-consuming. Indeed, determining an ideal "power-to-weight" ratio has slowed adoption in Haskell [Yorgey et al. 2012], and has led to repeated rewrites, and the abandonment of, Rust dependent type RFCs [2017]. This history suggests that language designers would benefit from an easier way to build and iterate on the design of dependently typed features and languages.

We present TURNSTILE+, a metalanguage for implementing typed—particularly dependently typed—languages. TURNSTILE+ supports terse mathematical notation, modular language feature implementation, and implicit handling of complex type system implementation patterns, including those for dependent types. These capabilities allow language designers to iterate quickly on the

Authors' addresses: Stephen Chang, Northeastern University, USA , PLT Group, USA, stchang@ccs.neu.edu; Michael Ballantyne, Northeastern University, USA , PLT Group, USA, mballantyne@ccs.neu.edu; Milo Turner, Northeastern University, USA, milo@ccs.neu.edu; William J. Bowman, University of British Columbia, Canada, wjb@williamjbowman.com.

2475-1421/2020/1-ART3
https://doi.org/10.1145/3371071

design of a new dependently typed feature by rapidly building modular and extensible prototype implementations. Or they may drastically reduce the size of their design space by tailoring the power of a type system to a specific DSL. Turnstile+'s ability to quickly create languages comes from its use of LISP and Scheme-style *macros*, which enable reusing a host language's infrastructure when building new languages. Turnstile+ is implemented in Racket [Felleisen et al. 2015], this paper's platform of choice, because it exposes more of the compiler for reuse than its predecessors [Flatt 2002, 2016; Flatt et al. 2012], and has a language-oriented focus [Felleisen et al. 2018].

Turnstile+ can also help advance the state-of-the-art for full-spectrum dependently typed languages like Coq or Agda, which often have powerful core type theories that are difficult to program directly. Thus, users rely on a variety of features layered on top of the core, from extensions for unification [McBride 2000], termination [Giménez 1995], or automation [Blanchette et al. 2016], to companion DSLs like tactic systems [Delahaye 2000; Gonthier and Mahboubi 2010; Ziliani et al. 2013] or pattern matchers [Coquand 1992; Norell 2007]. Unfortunately, such an ad-hoc system of interacting components can be difficult to modify, especially when third-party tools are involved. We claim our macro-based, language-oriented approach creates more extensible, linguistically-integrated components, and can thus help researchers more easily explore alternate designs.

The main technical contribution of our paper is the design and implementation of the Turnstile+ metalanguage, which is a complete rewrite of Chang et al. [2017]'s Turnstile. Turnstile showed that typed languages, instead of being built from scratch, could be implemented by adding some type checking code to macro definitions, and then reusing the infrastructure of the macro system—*e.g.*, its implementation of binding, pattern matching, environments, and transformations—for the rest of the type checker. Even better, organized in this way, a type checker implementation closely corresponds to its (algorithmic) specification, and language creators may exploit this correspondence by coding at the level of mathematical type rules. Turnstile+ represents a major research leap over its predecessor. Specifically, we solve the major challenges necessary to implement dependent types and their accompanying DSLs and extensions (which Turnstile could not support), while retaining the original abilities of Turnstile. For example, one considerable obstacle was the separation between the macro expansion phase and a program's runtime phase. Since dependently typed languages may evaluate expressions while type checking, checking dependent types with macros requires new macrology design patterns and abstractions for interleaving expansion, type checking, and evaluation. The following summarizes our key innovations.

- Turnstile+ demands a radically different API for implementing a language's types. It must be straightforward yet expressive enough to represent a range of constructs from base types, to binding forms like Π types, to datatype definition forms for indexed inductive type families.
- Turnstile+ includes an API for defining type-level computation, which we dub *normalization by macro expansion*. A programmer writes a reduction rule using syntax resembling familiar on-paper notation, and Turnstile+ generates a macro definition that performs the reduction during macro expansion. This allows easily implementing modular type-level evaluation.
- Turnstile+'s new type API adds a generic type operation interface, enabling modular implementation of features such as error messages, pattern matching, and resugaring. This is particularly important for implementing tools like tactic systems that inspect intermediate type checking steps and construct partial terms.
- Turnstile+'s core type checking infrastructure requires an overhaul, specifically with first-class type environments, in order to accommodate features like dependent binding structures of the shape $[x : \tau] \ldots$, *i.e.*, *telescopes* [de Bruijn 1991; McBride 2000].
- Relatedly, Turnstile+'s inference-rule syntax is extended so that operations over telescopes, or premises with references to telescopes, operate as "fold"s instead of as "map"s.

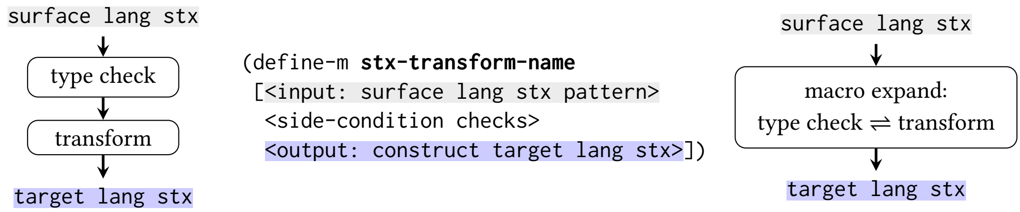

Fig. 1. (l) A typical type system implementation, (c) a macro definition, and (r) a macro-based type checker.

To evaluate our claim that Turnstile+ allows quickly and modularly iterating on dependently typed languages, and tailoring the power of a type system, we present a series of examples that range from "lightweight" to "full-spectrum", though we spend more time on the "full" end because it is more involved. Specifically we show how to create: a video-editing DSL with a Dependent ML-like type system; a full-spectrum dependently typed calculus à la Martin Löf; as well as one with inductive datatypes à la Dybjer [1994]. To show that our approach scales to a realistic language, we then turn our core inductive calculus into Cur, a prototype proof assistant with: recursive definitions and termination checking; unification and implicit arguments; and dependent pattern matching. To show that our system supports common extensions we create: a tactic system; a metaDSL to implement the tactic system; a system similar to Ott [Sewell et al. 2007] for writing language definitions; a library for proving theorems via an SMT solver; and a library that adds sized types [Abel 2010; Hughes et al. 1996]. Finally, to demonstrate that languages created with Turnstile+ are realistic to program with, we used Cur to work through a graduate-level semester's worth of examples—roughly volume 1 of the "Software Foundations" curriculum [Pierce et al. 2018].

2 CREATING A TYPED LANGUAGE (STLC) WITH RACKET AND TURNSTILE+

2.1 Interleaving Transformation and Type Checking with Macros

Macros are special compile-time functions that consume and produce *syntax objects* [Dybvig et al. 1992], *i.e.*, enhanced (with source location, binding structure, etc.) S-expression ASTs. Figure 1 (center) illustrates how they are suitable for implementing typed languages because the syntax transformation and side-condition components of a typical *macro definition* mirrors the checking and transformation (*e.g.*, to a lower-level core language) performed by type checkers (Figure 1 (left)). More specifically, a macro definition deconstructs its input via a *syntax pattern* (`gray` in this paper),[1] whose shape dictates the macro's usage syntax. This input is eventually transformed, after checking possible side-condition guards, into a macro's output, which is typically created with a quasiquotation *syntax template* (`blue`).

In a language with macros, a *macro expansion* compiler phase repeatedly rewrites a program's surface syntax according to all macro definitions, until no macro invocations remain. During expansion, if any of a macro's side-conditions are not satisfied, compilation fails with an error. By implementing type rules as these side-conditions, one may implement a type checker as macros. Since macro definitions are modular components, a macro-based type checker *interleaves* checking and transformation (Figure 1 (r)). This differs from the simple architecture depicted by Figure 1 (l), but it more closely follows the modular nature of a type system's rule specifications.

Figure 2 presents such rules for the simply typed λ-calculus, split into bidirectional "synthesize" ($\Rightarrow$)—the type is the output—and "check" ($\Leftarrow$)—the type is the input—variants. More specifically a

[1]To better communicate high-level concepts, some code may be stylized, *e.g.*, we may elide unimportant details, use abbreviations, `color`, or subscripts. For example, a `define-m` macro is shorthand for `define-syntax` and `syntax-parse` pattern matching. Thus, some code may not run as presented, but runnable examples for all code are in our artifact.

$$\textsc{App}{\Rightarrow}\ \frac{\Gamma \vdash f \gg \overline{f} \Rightarrow \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash e \gg \overline{e} \Leftarrow \tau_1}{\Gamma \vdash \mathrm{app} f e \gg \overline{\mathrm{app}}\, \overline{f}\, \overline{e} \Rightarrow \tau_2}$$

$$\textsc{Lam}{\Rightarrow}\ \frac{\Gamma, x \gg \overline{x} : \tau_1 \vdash e \gg \overline{e} \Rightarrow \tau_2 \quad \overline{x} \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda x : \tau_1 . e \gg \overline{\lambda} \overline{x}.\overline{e} \Rightarrow \tau_1 \rightarrow \tau_2}$$

$$\textsc{App}{\Leftarrow}\ \frac{\Gamma \vdash f \gg \overline{f} \Rightarrow \tau_1 \rightarrow \tau_2 \quad \tau_2 = \tau_0 \quad \Gamma \vdash e \gg \overline{e} \Leftarrow \tau_1}{\Gamma \vdash \mathrm{app} f e \gg \overline{\mathrm{app}}\, \overline{f}\, \overline{e} \Leftarrow \tau_0}$$

$$\textsc{Lam}{\Leftarrow}\ \frac{\Gamma, x \gg \overline{x} : \tau_1 \vdash e \gg \overline{e} \Leftarrow \tau_2 \quad \overline{x} \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \lambda x . e \gg \overline{\lambda} \overline{x}.\overline{e} \Leftarrow \tau_1 \rightarrow \tau_2}$$

$$\textsc{Var}{\Rightarrow}\ \frac{x \gg \overline{x} : \tau \in \Gamma}{\Gamma \vdash x \gg \overline{x} \Rightarrow \tau}$$

$$\textsc{Var}{\Leftarrow}\ \frac{x \gg \overline{x} : \tau \in \Gamma \quad \tau = \tau_0}{\Gamma \vdash x \gg \overline{x} \Leftarrow \tau_0}$$

Fig. 2. Bidirectional "check and transform" rules for the simply typed λ-calculus.

judgment $\Gamma \vdash e \gg \overline{e} \Rightarrow \tau$ reads "in context Γ, e transforms to $\overline{e}$ and has a type that matches pattern τ, and $\Gamma \vdash e \gg \overline{e} \Leftarrow \tau$ reads "in context Γ, e transforms to $\overline{e}$ and has type equal to τ. This kind of "check and transform" rule, *e.g.*, from Pierce and Turner [1998], is a common way to specify type systems. In our Figure 2 example, the "transform" part is a basic type erasure. Since tracking and manipulating the binders of a program are important for type checking, *e.g.*, when computing substitution or α-equality, the rules conservatively generate fresh binders for the target language to distinguish it from source language binders. This paper uses an "overline" convention to distinguish source and target language constructs. For example, a surface λ and (an explicitly-named) app function application transform to $\overline{\lambda}$ and $\overline{\mathrm{app}}$, respectively, in some (for now unspecified) target language. The next section shows how one might implement Figure 2's specification with macros.

2.2 From Specification To Implementation

Figure 3 presents two versions of a Racket *module* named STLC, each containing an implementation of Figure 2's specifications. A `#lang` at the top of a Racket module declares the language of that module's code; thus, Figure 3 (l) depicts RACKET code. This RACKET implementation of STLC consists of two (type checking) macro definitions, `#%app` and `λ`, each with two cases corresponding to analogous ⇒ and ⇐ rules from Figure 2.[2] The `#%app` macro's first case implements the App ⇐ rule; thus its input must additionally include an "expected" type from its context[3] which binds it to a `τ0` pattern variable.[4] The rest of the case follows straightforwardly from the rest of App ⇐. First, it computes the type of the applied function `f`[5], and its erasure, with a call to a `synth/≫` function (`synth/≫` and other helper functions are explained in Section 2.3). The pattern `(→ τ1 τ2)` constrains the type of `f` to be a function with one input and one output type (implementation of function types is explained in Section 3). If `f`'s type fails to match this pattern, type checking fails with an error. Next, the `#%app` macro must check that the output type of the function is equivalent to the expected type. Then the macro checks that the argument `e` has type `τ1` using function `check/≫`, which returns `ē`, the erasure of `e`. Finally, the macro emits an erased term `(#%app f̄ ē)`.

Like App ⇒ and App ⇐, the second `#%app` macro case is similar to the first. The difference is that there is no incoming "expected" type. Instead, the macro emits a second output, the type of the function application term, which is the `τ2` from the function type.

[2] We sometimes use square brackets, which are semantically equivalent to standard parentheses, to help readability.
[3] Like "this" from OOP, `this-stx` is a macro's syntax object input. It may have "expected" type information from the context.
[4] The `#:when` and `#:with` directives specify additional side conditions for a macro. Expansion may only proceed if predicates following `#:when` are true, and when the pattern after `#:with` matches the syntax computed by the subsequent expression.
[5] Here we use an explicit quasiquote constructor ``#` ``, which creates a syntax object according to the template that follows it.

```
#lang RACKET   (provide λ #%app)                        STLC
(define-m #%app
 [(#%app f e) #:when (has-expected-τ? this-stx)
  #:with τ0 (get-expected-τ this-stx)
  #:with [f (→ τ1 τ2)] (synth/≫ #`f)
  #:when (τ= #`τ2 #`τ0)
  #:with e (check/≫ #`e #`τ1)
  #`(#%app f e)]
 [(#%app f e)
  #:with [f (→ τ1 τ2)] (synth/≫ #`f)
  #:with e (check/≫ #`e #`τ1)
  (assign #`(#%app f e) #`τ2)])
(define-m λ
 [(λ x e) #:when (has-expected-τ? this-stx)
  #:with (→ τ1 τ2) (get-expected-τ this-stx)
  #:with [x e] (check/≫ #`e #`τ2 #:ctx #`[x : τ1])
  #`(λ (x) e)]
 [(λ [x : τ1] e)
  #:with [x e τ2] (synth/≫ #`e #:ctx #`[x : τ1])
  (assign #`(λ (x) e) #`(→ τ1 τ2))])
```

```
#lang TURNSTILE+ (provide λ #%app)                      STLC
(define-tyrule #%app
 [(#%app f e) ⇐ τ0 ≫
  [⊢ f ≫ f ⇒ (→ τ1 τ2)] [τ2 τ= τ0]
  [⊢ e ≫ e ⇐ τ1]
  --------------------------------
  [⊢ (#%app f e)]]
 [(#%app f e) ≫
  [⊢ f ≫ f ⇒ (→ τ1 τ2)]
  [⊢ e ≫ e ⇐ τ1]
  --------------------------------
  [⊢ (#%app f e) ⇒ τ2]])
(define-tyrule λ
 [(λ x e) ⇐ (→ τ1 τ2)≫
  [[x ≫ x : τ1] ⊢ e ≫ e ⇐ τ2]
  --------------------------------
  [⊢ (λ (x) e)]]
 [(λ [x : τ1] e) ≫
  [[x ≫ x : τ1] ⊢ e ≫ e ⇒ τ2]
  --------------------------------
  [⊢ (λ (x) e) ⇒ (→ τ1 τ2)]]])
```

Fig. 3. STLC implementation, (l) using Racket, and (r) using TURNSTILE+.

In any syntax template, identifiers bound by in-scope syntax patterns refer to the pieces of syntax matched in those patterns. For example, in `(#%app f e)`, `f` refers to `f` from a previous pattern. Identifiers not bound in a previous pattern, however, reference bindings from the rule's definition context. Thus `#%app` in the `#%app` macro's output is RACKET's function application form. (We also use the overline convention to distinguish untyped Racket forms from any typed counterparts we may define.) The `#%` prefix naming convention indicates an implicit form so programmers do not write `#%app` explicitly; instead, the macro expander automatically inserts it in front of applied functions. So the STLC module, by redefining `#%app` (and exporting it), *changes* the behavior of function application.

Figure 3 (l)'s λ macro definition implements the LAM rules from Figure 2, in much the same way as `#%app`. The interesting part of λ is that it calls `check/≫` and `synth/≫` with an extra context argument consisting of the binding `x` and its type τ. These functions return one more result as well, the new "erased" binder, which is used to construct the output term.

In the Racket ecosystem, *a module is a language implementation*, and the language's constructs are exactly the module's exports. Since STLC exports `#%app` and λ, writing `#lang STLC` at the top of a module means that the subsequent code is type checked by STLC's macros. Obviously, programmers cannot write any STLC code yet because we have not defined any types; in general, however, this ability to control language features—users of a particular `#lang` cannot arbitrarily access features from another one—is useful when implementing dependent types, where unrestrained language interoperation can accidentally introduce inconsistency in the underlying logic.

The close correspondence between Figure 2's specification and Figure 3 (l)'s RACKET implementation suggests that programmers could implement typed languages using syntax closer to Figure 2. Figure 3 (r) shows STLC, implemented with TURNSTILE+. Though the two sides of Figure 3 somewhat resemble each other—the same patterns and templates are just rearranged—the extra layer is important for usability and reasoning while programming. For example, the left uses explicit macro operations and thus report errors in this low-level language, *e.g.*, *"bad syntax"*. In contrast

TURNSTILE+

```
(define (assign e τ) (attach e 'type τ))
(define (synth/≫ e #:ctx [x : τ])
 (define x̄ (fresh))
 (define env
  (envadd-m (curr-env) #`x (assign x̄ #`τ)))
 (define ē (local-expand e env))
 (define τe (detach ē 'type))
 #`(#,x̄ #,ē #,τe))

(define (has-expected-τ? e) (has-prop? e 'exp))
(define (add-expected-τ e τ) (attach e 'exp τ))
(define (get-expected-τ e) (detach e 'exp))
(define (check/≫ e τ #:ctx ctx)
  (define e/exp (add-expected-τ e τ))
  #:with [x̄ ē τe] (synth/≫ e/exp #:ctx ctx)
  (if (τ= #`τe τ) #`(x̄ ē) (err "ty mismatch")))
```

Fig. 4. "Type systems as macros" core API.

the right embeds implicit patterns and templates in a language of type rules, and these abstractions enable more domain-appropriate errors, *e.g.*, *"type mismatch, expected X, got Y"*. Since TURNSTILE+ sugars the calls to `synth/≫` and `check/≫` functions with the syntax of the judgments from Figure 2, the rules are read similarly: the [⊢ e ≫ ē ⇒ τ] and [⊢ e ≫ ē ⇐ τ] "premise judgments" are read, respectively, "e transforms to a term matching pattern ē and has type matching pattern τ", and "e rewrites to a term matching pattern ē and has type τ". The "conclusion judgment" of a `define-tyrule`, is split into its input components at the top, corresponding to the macro's input, and the output components at the bottom, corresponding to the macro's outputs. This allows the rule implementation to be read like code, top to bottom.

2.3 Helper Functions

Despite their similarities, there are key differences between Figures 2 and 3(r): the latter has no VAR rule nor explicit Γ environments. To understand these discrepancies, we look at implementations of the `synth/≫`, `check/≫`, and `assign` functions, in Figure 4, which reveal how TURNSTILE+ reuses Racket's macro infrastructure to implement these missing parts, and to more generally facilitate type checking. These helper functions require only a few lower-level operations on syntax; thus our entire type checker is implemented "as macros". Specifically, the functions rely on three macro system features: (1) a programmatic way to add macro definitions to the macro environment, (2) a `local-expand` function that manually initiates macro expansion on a syntax object; and (3) a way of associating additional information (*e.g.*, types) with syntax objects; we use *syntax properties* which, via `attach` and `detach`, associate key-value pairs to syntax objects.

Individually, `assign` attaches a type to a term, at key *'type*. The "expected type" functions use the same API at key *'exp*. The `synth/≫` function consumes an expression e and an (optional) `ctx`—which has shape `[x : τ]`—representing bindings to add to a type environment. The `synth/≫` function first generates a fresh binding x̄ (corresponding to x̄ in figures 2 and 3). Then it creates a new macro environment instance, which extends the old one with a *new macro definition* named x. This new macro expands to x̄ and its type τ. In this way, the VAR rules from Figure 2 are implemented in exactly the same way as λ and `#%app`: as a macro that expands to the term and type outputs of the "type check and transform" relation. This also explains the lack of Γs in Figure 3; since our macro-based type checking *re-uses Racket's macro environment as the type environment*, scoping of type environment bindings is automatically handled by the macro expander. A programmer need only specify the *new* bindings like for λ. The environment structure with these new bindings are then passed to `local-expand`, which invokes the appropriate type checking macro for e. After expanding (*i.e.*, type checking and transforming) e, its *'type* syntax property is retrieved and `synth/≫` returns a triple, as syntax, containing the fresh name, the transformed ē, and its type (the quasiquotation `#,` escape operator allows splicing metalanguage terms).

The `check/≫` function first invokes `synth/≫` on term e, checks that the actual and expected type match using type-equality function `τ=`, and returns the expanded ē if successful. For this

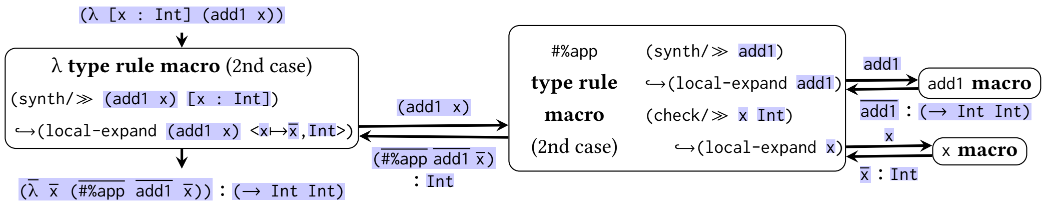

Fig. 5. Macro Call Graph For a Basic STLC Example

paper, we assume τ= (not shown) is syntactic equality up to α-equivalence; it is straightforward to implement directly on the concrete syntax, *i.e.*, we need not convert to an alternate representation like deBruijn indices, since syntax objects are already aware of the program's binding structure.

Figure 5 shows a macro call graph for an STLC example, `(λ [x : Int] (add1 x))`, where we assume Figure 3 is extended with arrow and `Int` types, and an `add1` primitive with type `(→ Int Int)`. Expansion (and type checking) begins with the λ macro, whose invocation calls the `synth/≫` function, which in turn calls `local-expand` with the lambda body, which invokes the next type checking macro, `#%app`. Return edges are marked with macro outputs, *i.e.*, expanded terms and their type (attached as a syntax property), so the example has final type `(→ Int Int)`.

3 LIGHTWEIGHT DEPENDENT TYPES, FOR VIDEO

Chang et al. [2017]'s original TURNSTILE could not handle dependent types since it assumes that terms and types are distinct. We introduce how TURNSTILE+ covers this deficit with TYPED VIDEO, a DSL with *indexed types*—"lightweight" dependent types in the style of Dependent ML [Xi 2007]—implemented "as macros". With indexed types, we can lift some terms (the index language) to the type level to express simple predicates about those terms. While Andersen et al. [2017] introduced TYPED VIDEO and briefly describe a few type rules, our work is the first to explain the underlying implementation details of such rules and their accompanying types. (As TYPED VIDEO is not our main focus, we do ignore unrelated parts of the language, *e.g.*, the details of constraint solving.)

TYPED VIDEO is a typed version of Andersen et al. [2017]'s VIDEO language, a DSL for editing videos that has been used to create the video proceedings of summer schools like OPLSS and conferences like POPL. TYPED VIDEO's indexed types statically rule out errors that arise when creating and combining video streams. More specifically, a VIDEO program manipulates *producers*—streams of data such as audio, video, or some combination thereof—cutting, splicing, and mixing them together into a final product. Since video editing is a multi-phase process, errors—*e.g.*, accidentally using more data than exists—often do not manifest until late in the editing process, usually during rendering, making them hard to find. To catch these problems earlier, TYPED VIDEO assigns producer values a `Prod` type, indexed by its length. This is an ideal type system since (even untyped) video editing already requires annotating many expressions with their length.

Below is a function that combines audio, video, and slides to create a conference talk video.

```
#lang TYPED/VIDEO                                                          VIDEO-PROG
(define (mk-conf-talk [n : Nat] [aud : (Prod n)] [vid : (Prod n)] [slide : (Prod n)])
                                                   -> (Prod (+ n 6)) #:when (> n 3)
  (playlist (img "conf-logo.png" #:len 6)
            (fade #:len 3)
            (overlay aud vid slide)))
```

The `mk-conf-talk` function consumes an integer length `n` and audio, video, and slide producers with types `(Prod n)`, meaning they must be at least `n` frames long. The function combines its inputs with `overlay`, and further adds a logo that fades into the main content. The function specifies an

TYPED/VIDEO

```
#lang Turnstile+
(define-type Nat : Type)
(define-type Prod : Nat -> Type)
(define-type →vid
      #:bind [X : Type] : Type -> Type)

(define-tyrule (#%app f e) ≫
  [⊢ f ≫ f̅ ⇒ (→vid [x̅ : τ1] τ2)]
  [⊢ e ≫ e̅ ⇐ τ1]
  ------------------------------------
  [⊢ (#%appvid f̅ e̅) ⇒ (subst e̅ x̅ τ2)])
```

```
(define-tyrule λ
 [(_ [x : τ1] e) ≫ ; Nat case
  [⊢ τ1 ≫ Nat ⇐ Type]
  [[x ≫ x̅ : Nat] ⊢ e ≫ e̅ ⇒ τ2]
  -----------------------------
  [⊢ (λ̅vid (x̅) e̅) ⇒ (→vid [x̅ : Nat] τ2)]]
 [(_ [x : τ1] e) ≫
  [⊢ τ1 ≫ τ̅1 ⇐ Type]
  [[x ≫ x̅ : τ̅1] ⊢ e ≫ e̅ ⇒ τ2] [⊢ τ2 ≫ _ ⇐ Type]
  -----------------------------
  [⊢ (λ̅vid (x̅) e̅) ⇒ (→vid [(fresh) : τ1] τ2)]])
```

Fig. 6. Typed Video type definitions, lambda, and function application rules.

additional constraint `(> n 3)`, to ensure that the inputs contain enough data to perform the fade transition. Finally, the output type specifies that the input is extended by 6 frames, to account for the added logo. The function is assigned a binding function type, where each argument is named:

```
(→vid [n : Nat] [a : (Prod n)] [v : (Prod n)] [sl : (Prod n)] (Prod (+ n 6)) #:when (> n 3))
```

The type of each argument may also reference the names preceding it, *i.e.*, it is a *telescope*. The next subsection shows how one may implement such a type with Turnstile+.

3.1 A Core for Typed Video, Now With Types

Figure 6 shows Typed Video's core, implemented with Turnstile+; it resembles STLC from Figure 3 (r) except extra `define-type` rules specify how to construct valid types. For example, the rule for `Nat` says that a `Nat` instance has "type" `Type`.[6] The `Prod` rule for producers says that constructing a `Prod` type requires a `Nat`, *i.e.*, a *a term*, argument. Finally, the $\rightarrow_{\texttt{vid}}$ specifies *binders*, which subsequent arguments may reference. The λ rule (we only show "synth" cases) shows how to create terms with $\rightarrow_{\texttt{vid}}$ types. In the first case, the parameter is a `Nat`, which may be referenced in both `e` *and its type*. Only `Nat` terms may be lifted, however, so a second λ case handles non-`Nat` parameters: its output $\rightarrow_{\texttt{vid}}$ is constructed with a `fresh` dummy parameter (note the quasiquoting), which is not referenced in $\overline{\tau}_2$. (The rule checks $\overline{\tau}_2$ a second time, to ensure that it does not reference `x`.) The `#%app` rule shows how `Nat` arguments are lifted to types: the conclusion replaces, in $\overline{\tau}_2$, references to the $\overline{x}$ binder with the application argument *term*.[7] Both rules transform into untyped Video terms.

3.2 Defining "Type" Rules For Types

Sections 3.2 to 3.4 explain how `define-type` is implemented. A `define-type` definition specifies a checking rule, for a type. We can implement such a rule with the previously seen `define-tyrule` because `define-tyrule`'s checking of syntax does not actually know the difference between "terms" or "types". Figure 7 shows what these explicit `define-tyrules` might look like for both plain and Typed Video's function types. The rule for a standard → checks that its input and output have type `Type`. But what should be the "transform" output of the rule? Typed λ transforms to a target language's $\overline{\lambda}$, but we have more freedom in choosing a type's underlying representation. Considering desired operations on types, however, reveals some criteria for such a representation. Specifically, type checking requires computing binding-aware operations like α-equality and capture-avoiding

[6] We assume `Type : Type` here. Section 5's Cur implements a hierarchy but this paper does not discuss these theoretical choices since they are largely orthogonal to our implementation techniques.

[7] This single-case rule, used in the rest of paper for brevity, combines the rule name and input pattern. For rules like these without a "check" clause, Turnstile+ uses a default "check" implemented with "synth" and τ=.

```
(struct ⇀ (in out))
(define-tyrule (→ τ1 τ2) ≫
  [⊢ τ1 ≫ τ̄1 ⇐ Type]
  [⊢ τ2 ≫ τ̄2 ⇐ Type]
  ----------
  [⊢ (#%app ⇀ τ̄1 τ̄2) ⇒ Type])
```

```
(struct ⇀vid (in out))
(define-tyrule (→vid [x : τ1] τ2) ≫
  [⊢ τ1 ≫ τ̄1 ⇐ Type]
  [[x ≫ x̄ : τ̄1] ⊢ τ2 ≫ τ̄2 ⇐ Type]
  ----------------------------
  [⊢ (#%app ⇀vid τ̄1 (λ̄ (x̄) τ̄2)) ⇒ Type])
```

Fig. 7. Explicit `define-tyrule` type rules for single-arity function types, (l) $\rightarrow$, (r) $\rightarrow_{\mathtt{vid}}$

```
(struct ⇀ (ins out))
(define-tyrule (→ τ1 ... τ2) ≫
  [⊢ τ1 ≫ τ̄1 ⇐ Type] ...
  [⊢ τ2 ≫ τ̄2 ⇐ Type]
  ----------
  [⊢ (#%app ⇀ (τ̄1 ...) τ̄2) ⇒ Type])
```

```
(struct ⇀vid (types))
(define-tyrule (→wrong [x : τ1] ... τ2) ≫
 [[x ≫ x̄ : τ̄1] ... ⊢ [τ1 ≫ τ̄1 ⇐ Type] ...]
 [[x ≫ x̄ : τ̄1] ... ⊢ [τ2 ≫ τ̄2 ⇐ Type]]
 ----------------------------
 [⊢ (#%app ⇀vid (λ (x̄ ...) τ̄1 ... τ̄2)) ⇒ Type])
(define-tyrule (→vid [x : τ1] ... τ2)
  [[x ≫ x̄ : τ1 ≫ τ̄1 ⇐ Type] ... ⊢ τ2 ≫ τ̄2 ⇐ Type]
  ----------------------------
  [⊢ (#%app ⇀vid (λ (x̄ ...) τ̄1 ... τ̄2)) ⇒ Type])
```

Fig. 8. Explicit `define-tyrule` type rules for multi-arity function types, (l) $\rightarrow$, (r) $\rightarrow_{\mathtt{vid}}$

substitution. Since syntax objects know a program's binding structure, using a binding-valid representation lets us exploit the macro system to implement such operations for free. Thus our criteria for a type representation: (1) uniquely identifies the type; (2) includes the arguments to the type constructor; and 3) respects hygiene, *i.e., it has a valid binding structure in the target language*. For the first two criteria, we define a (named) record $\overline{\rightarrow}$, declared with `struct`, and represent $\rightarrow$ with a *syntax object* that applies $\overline{\rightarrow}$ to the function type constructor's arguments, as seen in Figure 7 (l). For binding types like $\rightarrow_{\mathtt{vid}}$ in Figure 7 (r), to satisfy the hygiene criteria, its representation in the conclusion includes a $\overline{\lambda}$ that wraps and binds references to $\overline{x}$ in $\overline{\tau}_2$.

3.3 Type Checking Telescopes: A New Turnstile+ Premise

Suppose we want multi-arity function types. The $\rightarrow$ rule in Figure 8 (l) uses an ellipsis pattern `...`, which means "match the preceding pattern zero or more times", to specify multiple arguments. Any pattern variables in this preceding pattern, like τ_1, must be accompanied with another ellipsis when used in a syntax template, *e.g.*, τ_1`...`. Turnstile+ allows writing an ellipsis after a premise, which automatically inserts corresponding ellipses for *all* patterns and templates in that premise. For example the first premise in Figure 8 (l) checks that each type in τ_1`...` has type `Type`, and matches the expansion of those types to $\overline{\tau}_1$`...`.

The $\rightarrow_{\mathtt{wrong}}$ rule in Figure 8 (r) tries to use the same ellipsis pattern as the left, but this is the wrong binding structure because each τ_1 is checked in a context with *every* `x`, *including its own*. It's wrong because the ... usually means "map". But to type check the telescoping binding structure on the right, we need a "fold" operation that *interleaves binding with checking*. This means we need a new kind of premise, seen in the multi-arity $\rightarrow_{\mathtt{vid}}$ rule in Figure 8 (r), which is a corrected version of $\rightarrow_{\mathtt{wrong}}$. Specifically, a premise `[x ≫ x̄ : τ1 ≫ τ̄1 ⇐ Type]` ... checks each τ_1, but also names it so that subsequent type checking invoked by the ellipsis may reference the argument.

This new premise requires revising Figure 4's API to accommodate the fold operation; the new API is in Figure 9. The main function is now `expand/bind`, which consumes an identifier `x`, a syntax object `stx`, a `tag` at which to associate `x` with `stx` (e.g., *'type*), and a (macro) environment `env`. This new API is agnostic: we do not care whether `stx` is a type, term, or something else. The `expand/bind`

```
(define (expand/bind x stx tag env)        (define (expand/bind/check x stx tag stxval env)
  (define stx̅ (local-expand stx env))        (define env_new (expand/bind x stx tag env))
  (env-add env x tag stx̅))                   (if ([stxα=] (detach (lookup env_new x) tag) stxval)
(define (env-add env x tag stx̅)                  env_new
  (define x̅ (fresh))                             (err "ty mismatch")))
 (env-add-m env x (attach x̅ tag stx̅)))
```

Fig. 9. Type- and term-agnostic, revised version of the API from Figure 4; supports interleaved bind and check.

function expands `stx` in the context of `env`'s bindings, and then adds `x` to `env` as a "type rule" macro where `x` expands to a fresh $\overline{\texttt{x}}$, with `stx` attached at some `tag`. For rules that do not want to bind after checking, `expand/bind` can be called with a dummy `x` and `tag`. Folding "check" premises, however, would use `expand/bind/check`, which wraps `expand/bind`. It consumes an additional argument `stxval`, and "checks" that expanding `stx` results in a syntax object that has `tag` "equal" to `stxval`. Concretely, the premise of Figure 8 (r)'s $\rightarrow_{\texttt{vid}}$ rule would fold `expand/bind/check` over `x...`, τ_1`...`, and `Type...`, where each τ_1 is the `stx` argument and `Type` is `stxval`. The fold would accumulate the `env` argument which holds the resulting $\overline{\texttt{x}}$... and $\overline{\tau_1}$.... Where Figure 4 used τ=, Figure 9 uses $\boxed{\texttt{stx}\alpha\texttt{=}}$, which is an agnostic equality function. The box denotes an *interposition point*; Turnstile+ implements several of these overloadable hooks at key points to enable more extensible rules.

3.4 Putting it All Together

We need one more component for `define-type`. The rule outputs in Figure 8 are still somewhat arbitrary; a programmer should not need to know this underlying representation. Instead, they should use a type's *surface* syntax, as Figure 3 does when pattern matching with →. To enable this, we define "pattern macros" for each type, which are used exclusively in syntax pattern positions:

```
(define-pattern-m (→_vid [x : τ1] ... τ2) (#%app →̅_vid (λ̅ (x ...) τ1 ... τ2)))
```

This macro matches on, but hides, a type's internal representation. A pattern macro may have the same name as the type because its expansion occurs *before*, *i.e.*, in a different namespace from, a `define-tyrule`. This feature relieves Turnstile+ programmers from a heavy notational burden. With pattern macros, along with a `struct` declaration for the internal representation, and a `define-tyrule` implementing the type constructor rule we can implement `define-type`, which is just sugar for this collection of definitions.

3.5 Type-Level Computation

Since Typed Video types may contain expressions, α-equality alone no longer suffices. Thus, we also define a type normalization function, which is straightforward using the macro system's facilities for manipulating syntax. To install this normalization function, we do not need to change any rules; instead, we take advantage of another Turnstile+ interposition point, in `expand/bind`:

```
(define (expand/bind x stx tag env) (env-add env x tag ([norm] stx env)))
```

This new version modifies Figure 9's version by replacing `local-expand` with an overloadable $\boxed{\texttt{norm}}$ (that itself defaults to `local-expand`). This ensures that only normal forms are used when computing equality with $\boxed{\texttt{stx}\alpha\texttt{=}}$ in Figure 9. For Typed Video, we implement an interpreter for the index language. Figure 10 conveys the basic idea. It uses `define-`$\boxed{\texttt{norm}}$ to overload $\boxed{\texttt{norm}}$, which is implemented as a series of pattern-body cases. The first two cases match literal values. The third case matches on addition, recursively calling `norm` on the arguments. If evaluating those terms produce syntactic literal numbers, then the actual arithmetic operation is performed; otherwise, normalization produces a normalized addition syntax object. This fourth case is similar: if the input is a `Producer`, then its index is normalized. Finally, the last case leaves the type unchanged.

```
#lang Turnstile+                                                    TYPED/VIDEO
(define- norm
 [n #:when (nat-lit? n) n]                [b #:when (bool-lit? b) b]
 [(+ n m)    #:with n̄ (norm n)     #:with m̄ (norm m)
  (if (and (nat-lit? n̄) (nat-lit? m̄)) (+ (stx->lit n̄) (stx->lit m̄)) (+ n̄ m̄))]
 [(Producer n) (Producer (norm n))]    [other other])
```

Fig. 10. Excerpt of type-level evaluation in the Typed Video language.

```
#lang Turnstile+                                                    DEP-LANG
(define-type Π #:bind [X : Type] : Type -> Type)
(define-tyrule (#%app f e) ≫                 (define-tyrule (λ [x : τ] e) ≫
  [⊢ f ≫ f̄ ⇒ (Π [X̄ : τ̄₁] τ̄₂)]                   [⊢ τ ≫ τ̄ ⇐ Type]
  [⊢ e ≫ ē ⇐ τ̄₁]                                 [[x ≫ x̄ : τ̄] ⊢ e ≫ ē ⇒ τ̄₂]
  -----------------------------------            ------------------------------
  [⊢ (β f̄ ē) ⇒ (⇑ (subst ē X̄ τ̄₂))])               [⊢ (λ̄ (x̄) ē) ⇒ (Π [x̄ : τ̄] τ̄₂)])
(define-red β (#%app‾ (λ̄ (x) body) e) ~> (subst e x body))
```

Fig. 11. A full-spectrum dependently typed lambda calculus.

4 A DEPENDENTLY-TYPED CALCULUS

The approach to type-level computation for Section 3's Typed Video suffices for a simple index language but this approach is not extensible, *e.g.*, it breaks down for languages where introducing new datatypes is possible. This section presents a more general and extensible approach to adding type-level computation, which we dub *normalization by macro expansion* because each reduction rule is implemented as a separate macro. We explain using a calculus with full-spectrum dependent types, comparable to the Calculus of Constructions (CC) [Coquand and Huet 1988], in which there is no distinction between terms and types. We then extend this initial implementation with type schemas, à la Martin-Löf Type Theory [Martin-Löf 1975], and finally add inductive types, demonstrating that our approach scales to the calculi used in contemporary proof assistants. Further, each extension is modular: it only defines new constructs and does not modify prior code.

4.1 Defining Type-Level Reductions

This section introduces a Turnstile+ construct called `define-red`, for defining type-level computation. Figure 11 presents dep-lang, which uses `define-red` to define a β reduction rule by specifying a redex (as a syntax pattern) and a contractum (as a template). dep-lang also upgrades Figure 3's STLC by: (1) changing → to Π, the dependent function type whose output type can refer to its input; and (2) modifying λ and `#%app` to introduce and eliminate Π. Since the β macro is invoked in `#%app`'s output, type computation is interleaved with type checking.

A `define-red` definition internally defines a macro that rewrites redexes into contractums. Figure 12 sketches what a β macro might look like. If the first term is a $\overline{\lambda}$ (first case), occurrences of parameter $\overline{x}$ in the body are replaced with argument $\overline{e}$. This reduction may create more redexes in the contractum, *e.g.*, if the $\overline{e}$ argument is a function, so β's first case applies $\Uparrow_{v1}$, which "reflects" $\overline{\texttt{\#\%app}}$ references back to β, enabling further reductions. Otherwise (second case), the result of β is an unreduced $\overline{\texttt{\#\%app}}$ *neutral term*. This β conceptually captures "normalization by macro expansion", but it's still *not extensible* since $\Uparrow_{v1}$ would need to know about all possible reduction rules in advance.

Instead, Figure 13 defines a new ⇑, extensible via syntax properties. Instead of directly replacing $\overline{\texttt{\#\%app}}$, ⇑ traverses a piece of syntax and checks for a *'reflect-name* property. If it exists, its value is used as the reflected name. Correspondingly, `mk-reflected` creates such an annotated syntax,

```
(define-m β
  [((λ (x̄) body) ē) (⇑v1 (subst ē x̄ body))]
  [(f ē) (#%app f ē)])) ; neutral term

; ⇑v1: fn mapping #%app back to β,
; NOT extensible
(define (⇑v1 e) (subst β #%app e))
```

Fig. 12. Possible β reduction rule, implemented as a plain macro, but *not* extensible.

```
(define ⇑ ; extensible reflect fn
 [x #:when (and (id? x)(has-reflid? x))
  (detach x 'reflect-name)]
 [(e ...) ((⇑ e) ...)]
 [otherwise otherwise])
(define (mk-reflected placehold reflid)
  (attach placehold 'reflect-name reflid))

; examples:                                    TURNSTILE+
; ((mk-reflected #%app β) (λ x x) (λ x x))
;   = (#%app (λ x x) (λ x x))
; (⇑ ((mk-reflected #%app β) (λ x x) (λ x x)))
;   = (β (λ x x) (λ x x))
; (local-expand
;  (⇑ ((mk-reflected #%app β) (λ x x) (λ x x))))
;   = (λ x x)

(define-m define-red ; TURNSTILE+ form for defining reduction rules
 [(define-red red-name redex ~> contractum) (define-red red-name [redex ~> contractum])]
 [(define-red red-name [(placeholder redex-hd rst ...) ~> contractum] ...) ; multi-redex case
   ⌜(define-m red-name
       [(redex-hd rst ...) (⇑ contractum)] ...
   ⌞   [(e ...) ((mk-reflected placeholder red-name) e ...)]))⌟])
```

Fig. 13. Extensible TURNSTILE+ API for defining reduction rules, used to define β.

for use in unreducible neutral terms, by attaching a *'reflect-name* property to a given *placeholder* identifier. With our function application example, the `placeholder` is `#%app` and the `reflid` is β.

Finally, `define-red` in Figure 13 (bot) is a macro-defining macro that, given a redex pattern and contractum template, generates a macro with the necessary calls to ⇑ and `mk-reflected`. Thus multiple `define-red` declarations automatically cooperate with each other without knowing of each other's presence. The first case of `define-red` is shorthand for single-redex reduction rules. It recursively calls the multi-redex second case,[8] where the generated macro definition, `red-name`, is a generalized version of β from Figure 12. If this macro's inputs match the supplied redex pattern, it rewrites them to the specified contractum, letting Racket's macro patterns and templates automatically do the work. Otherwise, the result is an unreduced neutral term with the supplied placeholder at the head, marked with a *'reflect-name* property. Thus if later reductions transform the neutral term into a redex, ⇑ ensures that `red-name` is invoked again to reduce it.

4.2 A Little Sugar

Figure 11's DEP-LANG is roughly the Calculus of Constructions, which is not a real programming language yet. Fortunately, languages created with our approach are extensible via macros for free. We demonstrate this with first some small sugar extensions, which we then use to create larger extensions like type schemas and inductive datatypes. Figure 14 defines currying, multi-argument forms Π/c, λ/c, and app/c, which unroll into their univariate versions. Π/c also allows omitting the binder when there is no dependency; in this (third) case, a fresh identifier is used. This allows programmers to more concisely use Π like the simply typed → when appropriate; correspondingly it is doubly exported with this alternate → name. All these macros use a "dot" pattern, which matches a syntax object as a cons pair, split into its head and tail, *e.g.*, in `(Π b . rst)`, `b` is the first binder and `rst` is the rest of the type, which may include more binders. The DEP-LANG/SUGAR library exports these currying forms without their `/c` suffix, meaning users of the library will have the original constructs overloaded with these respective sugary forms.

[8] Since `define-red` is a macro-defining macro, it has nested syntax templates and patterns, making our usual coloring more difficult to see. To help, the outer syntax template in the second case is boxed with ⌜⌟ instead of the usual blue text color.

```
#lang DEP-LANG                                                    DEP-LANG/SUGAR
(provide/rename [Π/c Π] [Π/c →] [λ/c λ] [app/c #%app])
(define-m Π/c   [(_ e) e] [(_ [x : τ] . rst) (Π [x : τ] (Π/c . rst))]
                          [(_ τ . rst)       (Π [(fresh) : τ] (Π/c . rst))])
(define-m λ/c   [(_ e) e] [(_ x+τ . rst) (λ x+τ (λ/c . rst))])
(define-m app/c [(_ e) e] [(_ f e . rst) (app/c (#%app f e) . rst)])
```

Fig. 14. A DEP-LANG library that adds some syntactic sugar for currying.

```
#lang TURNSTILE+  (require DEP-LANG)  (provide Nat Z S elim_Nat #%datum)   DEP-LANG/NAT
(define-type Nat : Type)   (define-type Z : Nat)   (define-type S : Nat -> Nat)
(define-tyrule (elim_Nat n P mz ms) ≫
  [⊢ n ≫ n̅ ⇐ Nat]  ; target
  [⊢ P ≫ P̅ ⇐ (→ Nat Type)]  ; prop / motive
  [⊢ mz ≫ m̅z̅ ⇐ (P̅ Z)]  ; method for Z
  [⊢ ms ≫ m̅s̅ ⇐ (Π [k : Nat] (→ (P̅ k) (P̅ (S k))))]  ; method for S
  -----------
  [⊢ (eval_Nat n̅ P̅ m̅z̅ m̅s̅) ⇒ (P̅ n̅)])
(define-red eval_Nat
  [(eval_Nat Z P mz ms) ~> mz]
  [(eval_Nat (S k) P mz ms) ~> (ms k (eval_Nat k P mz ms))])
(define-m #%datum
  [(_ n) #:when (zero? n) Z]
  [(_ n) #:when (nat-lit? n) (S (#%datum (sub1 (stx->lit n))))]
  [(_ x) (#%datum x)])
```

Fig. 15. A DEP-LANG extension for natural numbers.

4.3 An Extension of Natural Numbers

To write interesting programs, DEP-LANG needs more data types. Thus we extend it—using already described tools and techniques—with a MLTT-style [Nordström et al. 1990] natural number type schema in Figure 15. Like all TURNSTILE+ constructs, `define-type` is not limited to defining "types" in the simply typed sense; instead, it's suitable for defining any "constructor" form. Thus this module uses it to define `Nat` *and* its introduction rules `Z` and `S`, corresponding to "zero" and "successor". The elimination form, `elim`$_{\texttt{Nat}}$, corresponds to a fold over the datatype. Following the terminology of McBride [2000], the form (`elim`$_{\texttt{Nat}}$ `n P mz ms`) takes *target* to eliminate `n`, a *motive* `P` that describes the return type of this form, and one *method* for each case of natural numbers: `mz` when `n` is zero and `ms` when `n` is a successor. Method `mz` must have type `(P Z)`, *i.e.*, the motive applied to zero, while `ms` must have type `(Π [k : Nat] (→ (P k) (P (S k))))`, which mirrors an induction proof: for any `k`, given a proof of `(P k)`, we show `(P (S k))`. Using `define-red`, we can define reduction rules for `elim`$_{\texttt{Nat}}$, one each for `Z` and `S`. Observe that the pattern macros `Z` and `S` (defined as part of `define-type`, explained in Section 3.4) help specify the reduction succinctly.

Finally, the `Nat` module overloads the meaning of *literal data* by extending `#%datum`, another Racket interposition point that wraps literals. With the new `#%datum`, users of the DEP-LANG/NAT can write numeric literals in place of the more cumbersome `Z` and `S` constructors. The last `#%datum` clause falls back to a core `#%datum`, making this extension compatible with others that might extend literal data. We could even support diamond extensions by importing *two* existing versions of `#%datum` (under different names) and use them in separate clauses of a new `#%datum`.

```
#lang Turnstile+  (require dep-lang)                              dep-lang/eq
                                              (define-tyrule (transport t P p w e)≫
(define-type = : [A : Type] [a : A] [b : A] -> Type)   [⊢ t ≫ t̄ ⇒ A] [⊢ w ≫ w̄ ⇐ A]
(define-type refl : [A : Type] [e : A] -> (= A e e))   [⊢ P ≫ P̄ ⇐ (→ A Type)]
(define-red eval= (eval= pt (refl A t)) ~> pt)         [⊢ p ≫ p̄ ⇐ (P̄ t̄)]
                                                       [⊢ e ≫ ē ⇐ (= A t̄ w̄)]
                                                       --------------
                                                       [⊢ (eval= p̄ ē) ⇒ (P̄ w̄)])
```

Fig. 16. A dep-lang extension for the equality type.

```
(define-tyrule (define-type name : [A : ϰ1] ... -> ϰ2) ≫              Turnstile+
  [[A ≫ Ā : ϰ1 ≫ ϰ̄1 ⇐ Type] ... ⊢ ϰ2 ≫ ϰ̄2 ⇐ Type]
  ---------------------------------
  [≻ ⌜(define-tyrule (name Ā ...) ≫⌝
       [⊢ Ā ≫ Ā̄ ⇐ ϰ̄1] ...
       ------------------
     ⌞[⊢ (name Ā̄ ...) ⇒ ϰ̄2])          ⌟] ; rest of the macro elided        )
```

Fig. 17. (Part of) the implementation of define-type, showing instantiation of telescopes.

Notably, we use `#lang Turnstile+` to implement this `Nat` module, not dep-lang, because (unlike Figure 14) this extension is *unsafe*. Ideally, the dep-lang trusted core should not have the ability to add new types and rules that could change the logic. Section 4.5 discusses managing extensions.

4.4 An Equality Type Extension, and Applying Telescopes

Figure 16 presents a module implementing an equality, or identity, type. The `transport` rule dictates that for any motive `P` such that `(P a)` holds, eliminating a proof that `a = b` allows concluding `(P b)`. The `eval=` reduction then rewrites to `pt` when the proof is a `refl` constructor. The `=` `define-type` declaration uses *telescopic* arguments. While Section 3.3 presented our technique of *checking* telescopes, *applying* telescopic constructors is equally tricky. This subsection addresses the latter with a novel, pattern-based substitution technique.

Figure 17 shows the relevant parts of `define-type`, which generates a `define-tyrule` that uses this technique. `define-type` first validates the `ϰ1...ϰ2` annotations supplied by the programmer, with the new premise syntax from Section 3.3. The conclusion (a ≻ form accommodates emitting top-level forms like definitions) produces the type rule for `name` types. The key is the reuse of the `Ā ...` pattern variables from the premises of the `define-type` as the pattern variables of the generated `define-tyrule`. When the `name` type constructor is called, `Ā` is bound to the arguments supplied to that constructor. Since `Ā` also binds references in `ϰ̄1...`, however, uses of `ϰ̄1...` in `name` *automatically* have `Ā` references replaced with the concrete arguments to the `name` type constructor, which is the desired behavior. In other words, we piggyback on substitutions that the macro system already performs with pattern variables in templates to instantiate type variables. Further, the technique is safe, *i.e.*, no variables are captured, thanks to hygiene.

4.5 INTERLUDE: Managing Extensions to a Trusted Core

Our macro-based approach gives library writers the same power as language implementers to add new types and rules. Of course, this is dangerous since they might implement rules incorrectly, changing the trusted core. Figure 18 (top) shows a blatant example. Using `assign`, it defines `false=true` as an arbitrary term with type `(= false true)`, rendering previously unprovable theorems, *e.g.*, that `(not x)` equals `x`, for all `x`, provable.

```
#lang DEP-LANG  (require dep-lang/bool dep-lang/eq)                BAD-DEP-LANG-PROG
(define false=true (assign (void)  (= false true))) ; should not be able to do this
(λ x (elim_Bool x (λ y (= (not y) y)) false=true (sym false=true)))
    ; proves (∀ [x : Bool] (= (not x) x))
```

```
#lang DEP-LANG (require dep-lang/bool dep-lang/eq unsafe/axiom)  DEP-LANG-PROG-AXIOMS
(define-axiom false=true (= false true))
(print-assumptions (λ x (elim_Bool x (λ y (= (not y) y)) false=true (sym false=true))))
    ; => Axioms used: false=true : (= false true)
```

```
#lang TURNSTILE+ (provide define-axiom print-assumptions)              UNSAFE/AXIOM
(define-m (define-axiom name τ)
  ⌞(define name (attach (assign (void)  τ) 'axiom name))⌟)
(define-m (print-assumptions e)
  (print "Axioms used: ~a" (find-axioms (local-expand #`e)))) ; scans e for axiom props
```

Fig. 18. (top) DEP-LANG program showing danger of extensions. (mid/bot) An axiom library to track extensions.

```
#lang TURNSTILE+   (require rosette) ; imports z3verify                    UNSAFE/Z3
(define-m (define-axiom/z3 name e τ)
  ⌞(define-m name #:when (z3verify τ) (attach (assign (void)  τ) 'axiom name 'z3 name))⌟)
```

```
#lang TURNSTILE+    (provide (rename [require/report require]))             DEP-LANG
(define (maybe-report-extensions! module-path)
  (when (not (equal? (get-lang module-path) DEP-LANG))
    (print "using extension:" module-path)))
(define-m (require/report module-path)
  (require module-path)    (maybe-report-extensions! module-path))
```

Fig. 19. (top) solver-aided axioms; (bot) overloading behavior of `require` and `provide`.

Fortunately, the ability of macros to control syntax can also help *tame* this power. This subsection shows several possibilities. Any extensions in the paper may use the following mechanisms. A first step is to exclude "unsafe" extension capabilities, like `assign`, from core DEP-LANG. Figure 18 (mid) shows the same program, but with "axioms" marked. Specifically, the program explicitly imports `unsafe/axiom` (Figure 18 (bot)), which has two constructs: `define-axiom` produces an `assign`, but tagged with an extra *'axiom* property; `print-assumptions` then scans a term for these marked subterms and reports them. Using these constructs, programmers can at least know when they are using unproven axioms. Proof assistants like Coq have a similar feature.

With our framework, we can do more; Figure 19 shows two possibilities. The first (top) is another axiom-definer library; it provides `define-axiom/z3`, which is like `define-axiom`, except it asks a Z3 solver to verify the theorem before accepting it. Since our extensions are linguistically supported, not third-party tools, we may use arbitrary Racket libraries; thus we call on the Rosette language [Torlak and Bodik 2014] to help translate to SMT-LIB terms. The resulting term is marked as *both 'axiom* and *'z3axiom*, enabling more fine-grained classification. (Of course, the solver must now be trusted. We are currently working to translate Z3 proof scripts back to DEP-LANG terms.)

A remaining loophole is that users must explicitly use these axiom libraries; they could just as easily use another library that does not mark axioms. Figure 19 (bot) shows one way to address this, by modifying a language's import mechanism. Specifically, `require/report` wraps `require` so it warns if an imported module is not implemented with the trusted DEP-LANG core (we can easily error instead of warning if we want a safe non-extensible language). If DEP-LANG uses `require/report` as its only import mechanism, users cannot circumvent it.

```
#lang Turnstile+                                                              DEP-IND-LANG
(define-tyrule (define-datatype T [A : τA] ... : τ  [C x+τ ... : τC] ...) ≫
 [[A ≫ Ā : τA ≫ τ̄A ⇐ Type]... [T ≫ T̄ : (Π [A : τA]... τ)] ⊢ [τ ≫ (Π [ī : τ̄i]... τ̄T) ⇐ Type]
                                   [(Π x+τ ... τC) ≫ (Π [i+x : τ̄1] ... (T τ̄2 ...)) ⇐ Type] ... ]
 #:with (((īx ... x̄rec) ...) ...) (find-recur T̄ (([i+x τ̄1] ...) ...))
 #:with ((τ̄2i ...) ...) (drop-params ((τ̄2 ...) ...))
 --------------------------
 [≻ ⌜(define-type T : [Ā : τ̄A] ... [ī : τ̄i] ... -> τ̄T) ; define the type                ⌝
     (define-type C : [Ā : τ̄A] ... [i+x : τ̄1] ... -> (T τ̄2 ...)) ... ; and the data constructors
     (define-tyrule (elimT v P m ...) ; define eliminator for T
       [⊢ v ≫ v̄ ⇒ (T Ā ... īv ...)] ; target
       [⊢ P ≫ P̄ ⇐ (Π [ī : τ̄i]... (→ (T Ā ... ī ...) Type))] ; motive
       [⊢ m ≫ m̄ ⇐ (Π [i+x : τ̄1]... (→ (P̄ īx... x̄rec)... (P̄ τ̄2i... (C Ā... i+x...))))] ...
       ----------------------
       [⊢ (evalT v̄ P̄ m̄ ...) ⇒ (P̄ īv ... v̄)])
     (define-red evalT ; define reduction rule for eliminator
   ⌞   [(elimT (C Ā ... i+x ...) P̄ m̄ ...) ~> (m̄ i+x ... (evalT x̄rec P̄ m̄ ...) ...)] ...)    ⌟])
```

Fig. 20. The `define-datatype` form allows DEP-IND-LANG programmers to define inductive datatypes.

4.6 Indexed Inductive Type Families

Instead of modifying the trusted core with type schemas, most proof assistants support safe extension with inductively-defined type families [Dybjer 1994]. In other words, the core is extended just once with a set of sound, general-purpose rules for defining new types. We straightforwardly add this capability to DEP-LANG, using the constructs we have already presented, to get DEP-IND-LANG. Specifically, Figure 20 presents `define-datatype`, which is based on Brady's presentation of TT [Brady 2005]. The complete implementation is mere tens of lines of code,[9] yet it makes DEP-IND-LANG comparable to the core of proof assistants like Coq, demonstrating that our macro-based approach scales to expressive type theories while maintaining convenient notation.

We use a concrete length-indexed list example to help explain `define-datatype`:

```
#lang DEP-IND-LANG                                                            LIST-PROG
(define-datatype Vec [A : Type] : (→ [i : Nat] Type)
  [nil  : (Vec A 0)]
  [cons [k : Nat] [x : A] [xs : (Vec A k)] : (Vec A (S k))])
```

The main source of complexity compared to previous type definitions is that indexed inductive types distinguish between *parameters* and *indices* (`A` and `i` in the figure). Parameters are invariant across the definition while indices may vary. Thus data constructor declarations (`nil` and `cons`) may reference parameters `A` from the type definition, but indices must be specific to each constructor.

Briefly, the `define-datatype` macro produces the following definitions: `define-types` to define the type and its data constructors; a `define-tyrule` elimination rule; and a `define-red` reduction rule for the eliminators. A line-by-line explanation follows.

- `(define-tyrule (define-datatype T [A : τA] ... : τ  [C x+τ ... : τC] ...) ≫`

This defines a new language construct named `define-datatype`. When used, `define-datatype` defines a type constructor named `T` that itself has type (Π `[A : τA] ...τ`). A colon distinguishes the parameters from the rest of the type and the τ part may include indices, as in the `Vec` example. The rest of this input pattern specifies the data constructors `C ...` that produce terms of type `T`; each `C` has type (Π `[A : τA]... x+τ ... τC`) where the `A ...` are the same parameters from `T`.

[9] Our goal here is to communicate the essence of the definition clearly; thus we do elide positivity checking and some other definitions that would clutter the code, but the actual implementation is not too much longer.

- [[A ≫ $\overline{A}$: τ_A ≫ $\overline{\tau}_A$ ⇐ Type] . . . [T ≫ $\overline{T}$: (Π [A : τ_A] . . . τ)] ⊢ [τ ≫ (Π [$\overline{i}$: $\overline{\tau}_i$] . . . $\overline{\tau}_T$) ⇐ Type]
  [(Π x+τ ... τ_C) ≫ (Π [$\overline{i+x}$: $\overline{\tau}_1$] ... (T $\overline{\tau}_2$ ...)) ⇐ Type] ...]

These premises validate the types supplied by a programmer writing a `define-datatype`. Since these types may recursively reference the type being defined, the type environment includes `T`. The pattern for the expansion of τ includes explicit $\overline{i}$ index pattern variables; similarly, the data constructor type patterns include $\overline{i+x}$ variables for the constructor arguments, which may include indices. Finally, the output of each data constructor must be of type `T`, applied to some $\overline{\tau}_2$ ....

- #:with ((($\overline{i}_x$ ... $\overline{x}_{rec}$) ...) ...) (find-recur $\overline{T}$ ((([$\overline{i+x}$ $\overline{\tau}_1$] ...) ...))
  #:with (($\overline{\tau}_{2i}$ ...) ...) (drop-params (($\overline{\tau}_2$ ...) ...))

These lines extract subcomponents of the datatype definition that are needed to define the eliminator and reduction rules. The first `#:with` finds the recursive arguments of the data constructors; that is, those arguments with type `T`, where each ($\overline{i}_x$ ...$\overline{x}_{rec}$) is a subset of its corresponding ($\overline{i+x}$ ...). The second `#:with` extracts the index arguments unique to each data constructor, *e.g.*, the index for `nil` and `cons` is `0` and `(S k)`, respectively.

- (define-type **T** : [$\overline{A}$: $\overline{\tau}_A$] ... [$\overline{i}$: $\overline{\tau}_i$] ... -> $\overline{\tau}_T$) ; define the type
  (define-type **C** : [$\overline{A}$: $\overline{\tau}_A$] ... [$\overline{i+x}$: $\overline{\tau}_1$] ... -> (T $\overline{\tau}_2$ ...)) ... ; and the data constructors

This defines type constructor `T` and data constructors `C`. Note that the latter includes parameters `A` that were not originally specified with the arguments of `C`.

- (define-tyrule (**elim_T** v P m ...) ; define eliminator for terms of type T
  [⊢ v ≫ $\overline{v}$ ⇒ (T $\overline{A}$ ... $\overline{i}_v$ ...)] ; target
  [⊢ P ≫ $\overline{P}$ ⇐ (Π [$\overline{i}$: $\overline{\tau}_i$] ... (→ (T $\overline{A}$ ... $\overline{i}$ ...) Type))] ; motive
  [⊢ m ≫ $\overline{m}$ ⇐ (Π [$\overline{i+x}$: $\overline{\tau}_1$] ... (→ ($\overline{P}$ $\overline{i}_x$ ... $\overline{x}_{rec}$) ... ($\overline{P}$ $\overline{\tau}_{2i}$ ... (C $\overline{A}$ ... $\overline{i+x}$ ...))))] ...

  [⊢ (eval_T $\overline{v}$ $\overline{P}$ $\overline{m}$ ...) ⇒ ($\overline{P}$ $\overline{i}_v$ ... $\overline{v}$)])

Defines an eliminator elim_T for terms of type `T`, which has three arguments: a target `v`, a motive `P`, and methods `m`, one for each `C`. This general eliminator definition almost exactly matches its theoretical presentation in [Brady 2005], again showing how Turnstile+ code closely matches its specification. The target `v` must have type `T`. In the pattern for `v`'s type, the reuse of pattern variables $\overline{A}$ from the premises to `define-datatype` uses the pattern-based type instantiation technique introduced in Section 4.4. Within this elimination rule, any other pattern variables from `define-datatype`'s input with references to $\overline{A}$, *e.g.*, $\overline{\tau}_i$ or $\overline{\tau}_1$, will automatically be instantiated with `v`'s parameters by the macro system. We *do not* use this technique, however for indices, which are new pattern variables $\overline{i}_v$. The motive `P` is a function that consumes indices and a value with type `T` at those indices, and returns a type for the result of elimination. As mentioned, $\overline{\tau}_i$ are the types of indexes, but automatically instantiated with the inferred concrete parameters of target `v`.

A call to the eliminator must include one method `m` for each constructor `C`. Each method consumes the same inputs as each `C`, as specified in the input to `define-datatype`, as well as one extra argument for each recursive $\overline{x}_{rec}$. These latter arguments represent recursive applications of the eliminators, so their types are specified by the motive `P`, *i.e.*, ($\overline{P}$ $\overline{i}_x$ . . . $\overline{x}_{rec}$). The type ($\overline{P}$ $\overline{\tau}_{2i}$ ... (C $\overline{A}$ ... $\overline{i+x}$ ...)) of each method's result is also determined by the motive, where the $\overline{\tau}_{2i}$ are the indices specific to each `C` constructor. Finally, the eliminator output calls reduction rule eval_T to reduce redexes where $\overline{v}$ is a fully-applied constructor. Its type is determined by the motive applied to `v` itself.

- (define-red **eval_T** ; define reduction rule for eliminator
  [(elim_T (C $\overline{A}$ ... $\overline{i+x}$ ...) $\overline{P}$ $\overline{m}$ ...) ~> ($\overline{m}$ $\overline{i+x}$ ... (eval_T $\overline{x}_{rec}$ $\overline{P}$ $\overline{m}$ ...) ...)] ...)])

This last definition is a `define-red` reduction rule (from Figure 13) consisting of a series of redexes, one for each constructor `C`. It states that elimination of a fully-applied constructor `C` reduces to an application of the method for that constructor, where the recursive arguments to the method

```
#lang TURNSTILE+    (require TURNSTILE+/unification)                    CUR
(define-tyrule (define-implicit name_abbrv = name #:omit n)
 [⊢ name ≫ _ ⇒ (Π [X : τ] ... τ_out)]
 #:with (τ_explicit ...) (stx-drop n (τ ...))   #:with (X_implicit ...) (stx-take n (X ...))
 ---------------
 [≻ ⌜(define-pattern-m (name_abbrv pat_explicit ...) (name X_implicit ... pat_explicit ...))
     (define-tyrule (name_abbrv arg_explicit ...) ⇐ τ_expect ≫
       [⊢ arg_explicit ≫ arg_explicit ⇒ τ_arg] ...
       #:with ϑ (unify ([τ_expect τ_out] [τ_arg τ_explicit] ...) ())
       #:with (arg_implicit ...) (lookup X_implicit ϑ)
       ------------------------
    ⌞ [≻ (name arg_implicit ... arg_explicit ...)])                        ⌟])
```

Fig. 21. CUR extension for declaring implicit arguments.

```
(define (unify constraints_τ=τ ϑ)                              TURNSTILE+/UNIFICATION
 (syntax-parse constraints_τ=τ
  [() ϑ] ; (1) base case     [([τ x] . rst) (unify ([x τ] . rst) ϑ)] ; (2) swap id to lhs
  [([x τ] . rst) #:when (in? x (dom ϑ)) (unify ([(lookup x ϑ) τ] . rst) ϑ)] ; (3) conflict
  [([x τ] . rst) #:when (not (occurs x τ)) ; (4) elim
   (unify (subst τ x rst) ([x τ] (subst_rng τ x ϑ)))]
  [([τ1 τ2] . rst) #:when (τ= τ1 τ2) (unify rst ϑ)] ; (5) delete
  [([(#%app C1 τ1...) (#%app C2 τ2...)] . rst) #:when (and (= C1 C2) (len= (τ1 ...) (τ2 ...)))
   (unify ([τ1 τ2] ... . rst) ϑ)] ; (6) decompose
  [([(λ x1 e1) (λ x2 e2)] . rst) (unify ([e1 (subst x1 x2 e2)] . rst) ϑ)] ; (7) HO case
  [([τ1 τ2] . rst) (tyerror "could not unify τ1 and τ2")])) ; (8) error
```

Fig. 22. TURNSTILE+ basic unification function.

are additional invocations of the eliminator on the recursive constructor arguments. The macro system's pattern language naturally associates each `C` with its method `m`, again resulting in a concise definition that matches what language designers write on paper.

5 FROM CALCULUS TO PROGRAMMING LANGUAGE: INTRODUCING CUR

To show that our approach to implementing dependent types scales to realistic languages, this section presents CUR, an extension of DEP-IND-LANG with features expected in such languages. Specifically, we add implicit arguments, pattern matching, and recursive top-level function definitions. To help implement these features, we show how a macro system makes it straightforward to implement operations like unification and features like generic methods for types in TURNSTILE+.

5.1 Implicit Arguments and Unification

Figure 21 shows `define-implicit`, a form for declaring implicit arguments (roughly like Coq's "Arguments" extension). It defines $\texttt{name}_{\texttt{abbrv}}$, which is equivalent to a given `name` function without its first `n` arguments. The `define-implicit` form emits two definitions; the first is a pattern macro that allows omitting the same arguments in patterns as well. The second, the new $\texttt{name}_{\texttt{abbrv}}$, relies on a TURNSTILE+ `unify` function to compute the omitted arguments. The `unify` function consumes constraints in the form of pairs of types that should be considered equal—here it's the types of its explicit arguments and the expected type of the whole term, paired with the analogous types from `name`'s original function type—and returns a set of substitutions ϑ for the variables in the types that would indeed make the types equal. The second `unify` argument is an initial (empty) substitution.

CUR

```
(define-tyrule (match e #:as x #:with-indx i... #:in τin #:return τout case...) ≫
  [⊢ e ≫ ē ⇐ τin]
  #:with def (get-datatype-def τin)    #:with elim-name (get-elim-name def)
  #:when (cases-complete? (case ...) def)
  [≻ (elim-name ē (λ i ... x τout) (case->method case def) ...)])
(define-m (case->method [(C x ...) body] def)
  #:with (xrec ...) (get-xrec (C x ...) def)
  (λ x ... xrec ... body))
```

Fig. 23. A pattern matcher for CUR.

TURNSTILE+/TYPEDEF

```
; (part of) TURNSTILE+ generic API
(define-tyrule (define-type tyname ... #:implements methname meth ...)  ;...
 ⌞(define tyname (hash-table methname meth ...))⌟)
(define-m (define-generic-type-method methname)
 ⌜(define-tyrule (methname τ) ≫                ⌝
   [⊢ τ ≫ (#%app tyname _...) ⇐ Type]
 ⌞ [≻ (dict-ref (get-dict tyname) methname)])⌟)
```

CUR

```
(define-generic-type-method get-datatype-def)  ; Use of TURNSTILE+ generic API
(define-tyrule (define-datatype ...)  ;...
 ⌞(define-type ... #:implements get-datatype-def (λ (ty) <the entire datatype def>))⌟)
```

Fig. 24. (top) TURNSTILE+'s generic interface for types, (bot) example use of the interface

Figure 22 shows a basic unification function as a TURNSTILE+ library; it is roughly the well-known Martelli and Montanari [1982] algorithm, with an extra higher-order case for binding forms. We show it to demonstrate how a macro system's syntax manipulation and handling of binding makes it straightforward to implement operations like unification, since the implementation cases more or less correspond to the algorithm's specification. The eight cases may be summarized as (in order of implementation): (1) base case, which returns the accumulated substitution ϑ; (2) swaps identifiers to the left side; (3) handles conflicting substitutions for a variable by adding a constraint; (4) eliminates a constraint by adding a substitution for x to ϑ, replacing x with τ in the existing substitutions and constraints; (5) drops a constraint if the types are equal; (6) decomposes constructor applications into constraints for its arguments; (7) a higher-order case that adds a new constraint for the bodies; and (8) an error case when a constraint has types that are not equal.

5.2 Dependent Pattern Matching and Generic Type Methods

Explicit eliminators are unwieldly to use; most programmers prefer pattern matching instead. Figure 23 sketches a dependent pattern matcher that is sugar for the underlying eliminator. (This basic matcher's goal is to illustrate TURNSTILE+'s generic interface. We are exploring more sophisticated translations, *e.g.*, Goguen et al. [2006].) The key to `match` is determining which eliminator to use. More specifically, for any given type, we need the original datatype definition. In the implementation of `match`, a generic `get-datatype-def` function returns this information. This function must behave differently depending on its argument type, however, and thus its implementation relies on a generic method interface for types in TURNSTILE+. Once `match` has the original datatype definition, it may generate the equivalent eliminator term by: (1) computing the eliminator name, (2) checking that the case patterns are complete, and (3) converting the case patterns to eliminator methods by adding the recursive arguments.

Figure 24 (top) sketches how Turnstile+'s generic type method interface works. First, `define-type` from section 3.4 is modified to additionally emit one more definition: a method table that is available during macro expansion. Then a programmer, using `define-generic-type-method`, may define a generic rule that, based on a given type, looks up the method in that type's table and dispatches to it. It can do this because the internal name of the type `tyname` is also the name of the method table. There's no name conflict because the names are bound at different phases of macro expansion.

Figure 24 (bot) shows a usage of this generic interface to implement `get-datatype-def`, the generic method used in Figure 23 that returns the original datatype definition. A programmer first declares the generic method with `define-generic-type-method`. Then, the `define-type` generated by `define-datatype` uses the extra `#:implements` option to define type-specific versions of the function. In this case, it just returns the entire input to `define-datatype`.

5.3 Recursive Function Definitions

Realistic programming languages allow programmers to write recursive top-level pattern-matching function definitions. Some languages might define such a construct as sugar over λs which, when combined with `match` from section 5.2, gets part of the way there, *e.g.*:

```
(define-tyrule (define_bad f [x : τ] ... : τ_out body) ≫
  [[x ≫ x̄ : τ̄] ... [f ≫ f̄ : (Π [x : τ] ... τ_out)] ⊢ body ≫ bodȳ ⇐ τ_out]
  ------------
  [≻ (define̅ f̄ (λ̅ (x̄ ...) bodȳ))])
```

Function `f` may call itself in its body because (1) `f` is added to the type environment, and (2) $\overline{\texttt{define}}$ in the target language allows recursive definitions. This falls short for Cur, however, for several reasons. First, it allows defining non-terminating functions like `(define loop [n : Nat] : Nat (loop n))`. Second, our reduction rules fire too eagerly so that even supposedly terminating functions will not terminate. For example, imagine the following `Nat` function:

```
(define_bad f [n : Nat] : Nat (match n [Z Z] [(S m) (f m)]))
; (f m) => (match m [Z Z] [(S m2) (f m2)])
;       => (match m [Z Z] [(S m2) (match m2 [Z Z] [(S m3) (f m3)])]) => ...
```

This function seems to terminate because `f` is only called recursively with a smaller argument. If `f` is sugar for a λ, however, the application `(f m)` in the body can always reduce with β and this will produce infinite applications of `f`. Instead, recursive functions should not reduce until they are applied to concrete constructors, *i.e.*, each pattern case should be a new reduction definition.

Figure 25 presents `def/rec/match`, which defines top-level pattern-matching recursive functions. It is simplified to one argument `x`, and non-dependent, non-nested patterns, and we do not show features like inaccessible patterns, in order to focus on the termination checking (we plan to eventually support Agda-style matching, *e.g.*, [Cockx et al. 2014; Coquand 1992]). The first `#:with` computes binders and their types from each pattern with a generic method `pat->ctxt`. The second creates the expected type of each case body by replacing `x` in τ_2 with the case pattern.

The third and fourth `#:with` tag marks some of the types with extra syntax properties, enabling termination checking. Specifically, the function's argument type τ is marked as a "rec arg" while the types returned by `pat->ctxt` are marked "rec ok" because they are "smaller" arguments. The last premise type checks the bodies of each case in a type context with: (1) the original input `x`, (2) binders $\texttt{x}_{\texttt{pat}}$ from the patterns, and (3) the function `f` itself. Note this `f`'s input type is now $\tau_{\texttt{rec}}$; similarly the type of $\texttt{x}_{\texttt{pat}}$ is the "marked" $\tau_{\texttt{recok}}$. But the original argument `x` has unmarked type τ. A `#:where` directive following the premise explains how these extra syntax properties are used. Specifically, the (overloadable) type equality function `τ=` *is changed for just the duration of checking the bodies*, in the following way. Two types are equal if they were equal with the old `τ=`.

```
(def (mk-recarg τ) (attach τ 'rec-arg true)) (def (mk-ok τ) (attach τ 'rec-ok true))  Cur
(define-tyrule (def/rec/match f [x : τ] : τ2 [pat bod] ...)
 #:with (([xpat τpat] ...) ...) ((pat->ctxt pat τ) ...) #:with (τout...) ((subst pat x τ2) ...)
 #:with τrec (mk-recarg τ)                        #:with ((τrecok ...) ...) (((mk-ok τpat) ...) ...)
 [ [x ≫ x̄ : τ] [xpat ≫ x̄pat : τrecok] ... [f ≫ f̄ : (Π [x : τrec] τ2)] ⊢ [bod ≫ b̄od ⇐ τout] ...
   #:where τ = (λ (τ1 τ2) (and (τ =OLD τ1 τ2)
                                (or (not (rec-arg? τ2)) (rec-ok? τ1) (err "nonterminate"))))]
 -------------
 [> ⌜(define-tyrule (f e) ≫                   ⌝
       [⊢ e ≫ ē ⇐ τ]
       ---------------
       [⊢ (f-eval ē) ⇒ τ2])
    ⌞(define-red f-eval [(f̄ pat) ~> b̄od] ...)⌟])
```

Fig. 25. A new recursive definition form for Cur, with termination checking.

Additionally, if the second type is a "rec arg", then the first type must be "rec ok", otherwise we raise a type error. In this way, only terminating recursive function definitions are allowed. Note that in a body, attempting to apply `f` to its original input `x` would fail because `x` is not marked "rec ok". A `def/rec/match` use emits two definitions, a `define-tyrule` defining `f` and a `define-red` specifying how to reduce applications of `f`. Specifically, an applied `f` is only reduced if its argument is a concrete value that matches one of the input patterns from its definition. In this way, recursive functions avoid the non-terminating reductions described earlier.

5.4 Sized Types: Implementing Auxiliary Type Systems

Section 5.3's termination check roughly corresponds to Giménez [1995]'s syntactic guards in many proof assistants. This conservative analysis, however, rejects some valid programs, *e.g.*, division via subtraction, because "non-increasingness" of arguments does not propgate through function calls:

```
(def/rec/match minus [n : Nat] [m : Nat] : Nat
  [Z _ => n] [_ Z => n]
  [(S n-1) (S m-1) => (minus n-1 m-1)])
(def/rec/match div_rejected [n : Nat] [m : Nat] : Nat
  [Z _ => n]
 [(S n-1) m => (S (div_rejected (minus n-1 m) m))]) ; syntactic termination check rejects recursive call
```

An alternative is sized types [Hughes et al. 1996], a more expressive termination analysis, but adding them is tricky due to its invasive nature on both a language's implementation and its usability. Concretely, they typically require threading an extra size argument into every type and term (see [Abel 2012]'s work, which inspired our ideas, for details) cluttering code and complicating operations like type equality. This section shows an experimental sized types library that potentially reaps the benefits while minimizing the negatives. More specifically, with macros and syntax properties, we can add "auxiliary" type systems, like sized types, that *operate in parallel* to the main one. The extra types are used when needed, *e.g.*, checking termination, and ignored when not, *e.g.*, type equality.

Figure 26 shows the essence of our library. "Sized" types are plain types annotated with a "sz" property, where the size property can be either an arbitrary identifier `i`, or `(< sz)` where `sz` is another size property. With the extensibility afforded by macros, we only need to overload a few features. Specifically, the library consists of two main forms: `lift-datatype`, which lifts an existing datatype definition to be sized, and `def/rec/match`$_{sz}$, which reimplements `def/rec/match` from Figure 25 except with sized types for termination analysis. Here is the previous `div` example:

```
(define inc-sz [(< i) i] [else (fresh)])    (define (dec-sz sz) (< sz))   CUR/SIZEDTYPES
(define (get-sz τ) (or (detach τ 'sz) INF)) (define (add-sz τ sz) (attach τ 'sz sz))

(define-tyrule (lift-datatype TY)
 #:with df (get-datatype-def TY) #:with (C ...) (get-datacons df) #:with (τC ...) (get-τC df)
 [≻ ⌜(define-tyrule (Csz arg) ≫                                          ⌝
       #:with sz (inc-sz (get-sz arg))
        --------------
        [⊢ (C arg) ⇒ (add-sz τC sz)]) ...
      (define-instance Csz (pat->ctxt pat ty)
        #:with ([x τ] ...) (pat->ctx (subst C Csz pat) ty)
    ⌞   ([x (dec-sz τ)] ...)) ...                                        ⌟])

(define-tyrule (def/rec/matchsz f [x : τ #:sz i] : τout #:sz j [pat bod] ...)
 #:with τi (add-sz τ i)     #:with τout/j (add-sz τout j)
 #:with (([xpat τpat] ...) ...) ((pat->ctxt pat τi) ...) ; τpat ... has size (< i)
 #:with τ<i (add-sz τ (< i))     #:with τout/<j (add-sz τout (< j))
 [ [x ≫ x̄ : τi] [xpat ≫ x̄pat : τpat] ... [f ≫ f̄ : (Π [x : τ<i] τout/<j)] ⊢ [bod ≫ b̄od ⇐ τout/j] ...
   #:where [τ =] (λ̄ (τ1 τ2) (and ([τ =OLD] τ1 τ2) (sz-ok? (get-sz τ1) (get-sz τ2)))) ]
 -------------
 [≻ ⌜(define-tyrule (f e) ≫                                          ⌝
       [⊢ e ≫ ē ⇐ τ] #:with i (get-sz e)
       ----------------
       [⊢ (f-eval ē) ⇒ (add-sz τout j)]])
     ⌞(define-red f-eval [(f̄ pat) ~> b̄od] ...)⌟])

(define (sz-ok? sz1 sz2)  ; true when sz1 <= sz2
 (or (INF? sz2) (syntax-parse (sz1 sz2)
                 [(x y) (and (id? x) (id? y) (id=? x y))]   [((< x) (< y)) (sz-ok? x y)]
                 [((< x) y) (sz-ok? x y)]              [else (err "non-terminating!")] )))
```

Fig. 26. A library for Cur that adds sized types.

```
#lang Cur    (require cur/sizedtypes)    (lift-datatype Nat)
(def/rec/matchsz minussz [n : Nat #:sz i] [m : Nat] : Nat #:sz i ; minussz is non-increasing in size
  [Zsz _ => n]    [_ Zsz => n]
  [(Ssz n-1) (Ssz m-1) => (minussz n-1 m-1)])
(def/rec/matchsz divsz [n : Nat #:sz i] [m : Nat] : Nat #:sz i
  [Zsz _ => n]
  [(Ssz n-1) m => (Ssz (divsz (minussz n-1 m) m))]) ; sized type termination accepts
```

`lift-datatype` takes a type `TY`, previously defined with `define-datatype`, and defines sized wrappers C_{sz} for each unsized constructor `C`. To simplify understanding, we show each `C` with only one argument; an actual implementation would have to choose the "decreasing" argument. Each C_{sz} constructor adds a size to the type of an applied `C` term that is the size of its argument "incremented", where incrementing a size either removes a < or generates a fresh id. Dually, `lift-datatype` overloads the generic `pat->ctxt` for C_{sz} so that the pattern binders have type that is "decremented". This new `pat->ctxt` is used by the new `def/rec/match`$_{sz}$.

Except for the different termination analysis, the new `def/rec/match`$_{sz}$ is roughly the same as its predecessor. To implement termination via sized types, the new definition form requires size annotations on its types, which are then used while type checking the body of each case. Observe that when type checking the bodies, the type for `f` in the type environment requires an argument that is sized less than the original argument size `i`. This smaller argument will typically come from

```
#lang cur (require cur/olly)   OLLY-PROG
(define-language stlc #:vars (x)
 #:coq-out "stlc.v" #:latex-out "stlc.tex"
 val (v) ::= true false unit
 type (A B) ::= boolty unitty (-> A B) (* A A)
 trm (e) ::= x v (lm (#:bind x : A) e) (ap e e)
  (cons e e) (let (#:bind x #:bind x) = e in e))
```

```
(define-datatype stlc-trm : Type
 (Var->stlc-trm Var : stlc-trm)
 (stlc-val->stlc-trm stlc-value : stlc-trm)
 (stlc-lm Var stlc-type stlc-trm : stlc-trm)
 (stlc-ap stlc-trm stlc-trm : stlc-trm)
 (stlc-cons stlc-trm stlc-trm : stlc-trm)
 (stlc-let Var Var stlc-trm stlc-trm : stlc-trm))
```

Fig. 27. (l) STLC with Olly, a Cur notation extension; (r) Olly-generated Cur datatype

the result of the new `pat->ctxt` (generated by `lift-datatype`). Like previous `def/rec/match`, $\boxed{\tau =}$ is overloaded, this time to ensure that sized types satisfy a `sz-ok?` predicate, which enforces that its first argument has size less than or equal to its second. The `sz-ok?` function special-cases an `INF` size, which is assigned to unlifted types, for when we do not care about sizes. Most importantly, the size annotation on the output, which may reference sizes of the input arguments, allows declaring that functions like `minus` have non-increasing size. This allows size information to propagate across function calls to enable `div`, and even higher-order cases like "rose trees" (see [Abel 2010]).

6 COMPANION DSLS FOR A PROOF ASSISTANT

Even with Section 5's extensions, it is still tedious to program and prove with Cur. To make proving practical, proof assistants typically layer companion DSLs on top of their core. By building with macros from the beginning, we already have a framework in which both language implementers and users can easily build such DSLs. They may even build metaDSLs to build their DSLs, as advocated by language-oriented programming. Best of all, any new DSLs are linguistically integrated with Cur, instead of operating as third-party preprocessors. This section presents three DSLs: Olly, for modeling programming languages; ntac, a tactic language for scripting proofs; and metantac, a metaDSL used to implement ntac. All these DSLs elaborate to core Cur before type checking; thus we can extend the functionality of our language yet keep the trusted base small.

Olly is an Ott-inspired [Sewell et al. 2007] DSL for modeling programming languages in Cur. Specifically, programmers write BNF or inference rule notation to specify language syntax and relations, respectively, and Olly generates the Cur inductive type definitions; LaTeX or Coq extraction is also supported. Figure 27 (l) shows the STLC in BNF using Olly. Optional `#:bind` annotations specify binding positions in the grammar; here `cons` creates pairs and `let` eliminates them, thus the latter binds two names. Olly generates an inductive datatype for each non-terminal in the grammar; Figure 27 (r) shows `trm`, whose constructor names are derived from the specification. Extra constructors, *e.g.*, `Var->stlc-trm`, allow converting from the other non-terminals. Internally, `define-language` uses an intermediate data structure, which is converted to Cur, Coq, LaTeX, and other outputs. Unlike Ott and other external tools, Olly is a user-written library and is supported linguistically; thus programmers may use Olly forms alongside normal Cur code rather than switch to external tools, demonstrating how our macros-based approach supports tailoring all aspects of a proof assistant to specific domain, from the object theory to the syntax.

Tactic systems are a popular tool to enable interactive, command-based construction of proof terms in proof assistants and our macro-based approach naturally provides all the capabilities required to build one: pre-type-checking general purpose computation, traversal and pattern matching of language terms, interesting elaboration system data structures for manipulating proof states, an API to the object language to type check and evaluate terms while constructing proofs, interactivity, and syntactic integration into the language. Even better, we may abstract over these low-level features with a meta DSL, to implement the tactics concisely and intuitively.

We present ntac, a tactic language for Cur. It tracks intermediate hole-embedded proof terms, and subgoals and contexts corresponding to those holes, as a tree. Further, a zipper navigates and

```
#lang METANTAC ; define-tactic usage pattern
(define-tactic tactic-name
 [<usage pat> #:current-goal <goal pat>
  ; ...
  ; implicit bindings: $ctx, $ptz, $pt, $goal
  ; ...
  (↪ <new partial term with ?HOLEs>
   #:where
    [x : τ ... ⊢ ?HOLE1 : <subgoal1>] ...)]
 ...)
```

```
#lang METANTAC                                            NTAC
(define-tactic intro
 [(_ y) #:current-goal (Π (x : P) τ)
  (↪ (λ (y : P) ?H)
    #:where y : P ⊢ ?H : (subst y x τ))]
 [_ #:current-goal (Π (x : P) τ)
  (↪ (λ (x : P) ?H) #:where x : P ⊢ ?H : τ)])
(define-tactic assumption
 [_ (↪ (ctx-find $ctx (λ t (τ= t $goal))
          #:fail "no assumption $goal"))])
```

```
(define-tactic (inversion H #:with-names H_0 ...)    #:with τ_H (typeof H)      NTAC
 #:with ((A ...) (i ...) [C x ... x_rec ... : τ_C] ...) (get-datatype-def τ_H)
 (↪ (elim H (λ i ... H $goal) (λ x ... x_rec ... $pf) ...) #:with-subgoals
  ((stx-parse (unify+prove (get-indxs τ_H) (get-indxs τ_C))
    [([=pf : =thm] ...) (↪ ((λ [H0 : =thm] ... ?H0) =pf ...) #:where [H0 : =thm] ... ⊢ ?H0 : $goal)]
    [pf-of-False ; unify fail (↪ (elim_False pf-of-False (λ _ $goal)))]) ...)
```

Fig. 28. (l) `define-tactic` usage; (r) implementation of `intro` and `assumption`; (bot) `inversion` tactic

focuses on subgoals in this proof tree. Concretely, an NTAC tactic is a macro that when invoked, produces a function that transforms zipper data structure instances. Here is a trivial NTAC proof:

```
#lang CUR   (require cur/ntac)                                   EXAMPLE-THEOREM
(define id (ntac (∀ (A : Type) (a : A) A) (intros A a) assumption))
```

The first argument to `ntac` is a goal theorem; the rest are tactic invocations, *i.e.*, a script that builds the proof term. When invoked, `ntac` builds a tree node with the given goal and a hole term, and creates a zipper with that initial node as the focus. Each subsequent tactic invocation transforms the zipper and proof tree with nodes that gradually fill the hole(s). After the script completes, the holeless tree is converted to a complete proof term, which is the result of the `ntac` call. Thus, an `ntac` invocation may be used in any expression position, *e.g.*, bound to the name `id` using `define`.

A CUR tactic manipulates the proof zipper but a tactic programmer should not have to explicitly manage this proof state. Figure 28 (l) shows `define-tactic`, from a METANTAC language for writing tactics, which abstracts over low-level details. Specifically, a `define-tactic` definition consists of a series of cases, each starting with a pattern dictating usage syntax of the tactic, and an optional pattern matching the current goal. The body of each case must return a term, possibly with holes, whose type matches the current goal. Programmers use ↪ to construct this partial term, where optional `#:where` declarations, of shape `[x : τ] ... ⊢ ?HOLE : <subgoal>`, specify new subgoals to prove, each corresponding to a `?HOLE` name referenced in the first argument of ↪. They may also use implicitly bound variables if they wish to directly access parts of the proof state, *e.g.*, `$goal`, `$ctx`, `$pt`, `$ptz`, for the goal, context, proof tree, and zipper instance, respectively.

Figure 28 (r) shows the implementation of the two tactics used in the `id` theorem above. The `intro` tactic (only single-variable cases are shown) has two cases; the first is invoked when given an identifier `y`, when the goal has shape `(Π [x : P] τ)`. It fills the current hole with a λ binding `[y : P]`, which has a new hole `?H` in its body. It then specifies that a new subgoal for `?H`, in context where `y` has type `P`, is `τ` except with `x` replaced by the argument `y`. The second `intro` case has no argument; instead, it directly uses `x` from the goal. The `assumption` tactic also has no argument; it searches the current context, bound to `$ctx`, for a variable with type matching the current goal. If it's successful, it fills the current hole with the variable; otherwise it raises an error. No `#:where` argument for ↪ is necessary here because the resulting proof term has no holes.

METANTAC also supports writing tactics that may generate an unknown number of subgoals; *e.g.*, Figure 28 (bot) sketches `inversion`, which for an existing theorem `H`, generates equalities that must also be true based on the injectivity of constructors. For example, inverting `(= (S x) (S y))` produces `(= x y)`. The `inversion` tactic uses `unify+prove`, which performs specialization by unification [Goguen et al. 2006], computing either a series of equalities and their proofs, or a proof of `False`. If the former, the current proof state is extended with the new equalities; if the latter, the current goal is immediately proved. Specifically, `inversion` unifies the indices of $\tau_{\mathtt{H}}$, `H`'s type, with the indices of the result of each constructor `C` of $\tau_{\mathtt{H}}$, creating a proof subnode for each `C`. The result of `inversion` is an `elim` term for `H`, where the bodies of the methods are the results of the given subgoals, referenced with an implicit `$pf` variable.

By implementing tactics as functions, *tacticals*, *i.e.*, tactic combinators, are straightforward. For example, here is `try`, which takes a sequence of tactics but backtracks if any of them fail:

```
(define-tactical try                                                        NTAC
  [(_ t ...) (with-err-handler (λ (e) $ptz) ((compose (reverse (list t ...))) $ptz))])
```

It reverses its given tactics and applies them with with `compose`, which applies its rightmost argument first. If any of the tactics errors, then `try` reverts to the original proof state `$ptz`.

Macros also enable more flexible control of the surface syntax, even allowing hybrid tactic-tacticals. For example, here is a skeleton of an `induction` tactic with two cases:

```
(define-tactic induction                                                    NTAC
  [(_ H #:as ((x ...) ...)) ;...]
  [(_ H [(C x ...) #:subgoal-is subg tactic ...] ...) ;...])
```

The first is similar to systems like Coq, where the user supplies identifiers that will bind the data constructor arguments in each case. Programmers, however, can find it hard to read this kind of command. The second case enables a slightly more "declarative" usage: each data constructor case is named, and thus may appear in arbitrary order; the subgoals are explicit and checked, making the script easier to follow; and the tactics for each case are grouped, giving the proof more structure.

We can even equip user-defined tactics with features like interactivity:

```
(define-tactic interactive
 [_ (print $pt)
  (match (read-syntax)
   [(quit) $ptz]
   [a-tactic (interactive (a-tactic $ptz))])])
(ntac (∀ (A : Type) (a : A) A) interactive)
```

```
goal 1 of 1: (∀ (A : Type) (a : A) A)
> (by-intros A a)
      ctx:  A : Type   a : A   (step #1)
            ---------------
curr goal:  A
> by-assumption
Proof complete (2 steps)
> (quit) ; script: (intro A a) assumption
```

Specifically, the `interactive` tactic uses `print` to display the proof state, then starts a read-eval-print-loop (REPL). The right side shows an example interactive session. The REPL repeatedly reads in a command and runs it; when it sees `quit`, it prints the complete proof script and evaluates to the resulting proof term. We conjecture we could also embed NTAC with IDEs like emacs or DrRacket, perhaps using the techniques of Korkut and Christiansen [2018], for even better interactivity.

Programming with CUR and NTAC To demonstrate that one may usefully program with the languages we create, we implemented a large test suite. In particular, we spent one year using CUR and NTAC to study the *Software Foundations* curriculum (vol 1). Since it targets novices, its examples cover a wide breadth of features and is thus a convenient way to stress-test the flexibility of our macros-based approach to implementing dependent types. This table summarizes our test suite:

OLLY	59	DEP-LANG	272	DEP-IND-LANG	2914	TYPED/VIDEO	721
NTAC (sf vol 1)	8045	sized types	239	axioms	98		
sugar	58	patterns and def	122	solver	202	**total**: ~13.3k LoC	

7 RELATED WORK

There are many tutorials on **implementing dependent types** [Altenkirch et al. 2010; Augustsson 2007; Bauer 2012; Löh et al. 2010; Weirich 2014]. They typically start from scratch, however, *e.g.*, they manually manage type environments and rely on deBruijn indices to compute α-equality. They also often do not include practical features such as user-defined inductive datatypes, nor are they easily extensible with sugar, interactivity, or companion DSLs that programmers typically need to use with their dependently typed language. In contrast, our macros-based approach enables rapid creation of a core dependently typed language, and scales to a full proof assistant.

Extending proof assistants, particularly via metaprogramming, is an active area of research [Christiansen and Brady 2016; Devriese and Piessens 2013; Ebner et al. 2017]. For some languages, however, this requires extending the core [Brady and Hammond 2006]. Other languages like Coq require writing extensions in a less integrated manner, *e.g.*, programming plugins with OCaml and then linking it with other language binaries. We present an alternative, linguistically integrated approach to extensibility, using the macro system inherited from the host language.

New tactic languages continue to make proof assistants easier to use [Gonthier and Mahboubi 2010; Gonthier et al. 2011; Krebbers et al. 2017; Malecha and Bengtson 2016]. This suggests that (1) the ability to create a variety of tactic languages is critical, and (2) that linguistic support for creation of such DSLs would be well received. While we have yet to conduct a thorough comparison of all tactic languages and their implementations, we conjecture that our macros-based approach could accommodate many of them in a convenient manner. For example, there has been recent exploration of typed tactic languages Beluga [Pientka 2008], Mtac [Ziliani et al. 2013], and VeriML [Stampoulis and Shao 2010]. We conjecture that it would be straightforward to add a typed tactic language to Cur using our macros-based approach. This could be done either by utilizing Turnstile+, or using Cur's reflection API to use Cur as it's own meta-language, following work in Lean [Ebner et al. 2017], Idris [Christiansen and Brady 2016], Agda [Norell 2007], or Coq [Anand et al. 2018].

8 FUTURE WORK

In addition to exploring typed tactics and extensions like automation, we will also continue adding features and improving various aspects of our framework. For example, if `define-red` considered type information in addition to a redex pattern, it would enable implementing type-directed equality rules like η. Another potential improvement involves preventing abstraction leaks due to the interleaving of checking and expansion. For example, Cur and ntac must resugar during interactive proofs to avoid exposing users to elaborated syntax. The current approach is ad-hoc, however, recent advances [Pombrio and Krishnamurthi 2015] could help. Another solution could be to implement a *domain-specific core target language*, thus avoiding resugaring altogether. Such a feature would also improve the performance of type checking by culling extraneous expansion steps. We are exploring such enhancements, which possibly include changes to the macro expander itself, in order to further advance type checking with macros.

9 CONCLUSION

To fully realize the benefits of dependent types, programmers should be able to quickly develop dependently typed DSLs with the right power for a domain, and rapidly iterate on new dependently typed language features. Further, such languages should be easily extensible with new notation or companion DSLs that may be required for practical use cases. We have demonstrated that a macros-based approach to building dependently typed languages and features satisfies this criteria.

ACKNOWLEDGMENTS

We acknowledge the support of the NSERC grant RGPIN-2019-04207, and NSF grants 1823244 and 1518844. Cette recherche a été financée par le CRSNG, numéro de référence RGPIN-2019-04207.

REFERENCES

2017. RFC: The pi type trilogy. https://github.com/rust-lang/rfcs/issues/1930

Andreas Abel. 2010. MiniAgda: Integrating Sized and Dependent Types. In *PAR (EPTCS)*, Ana Bove, Ekaterina Komendantskaya, and Milad Niqui (Eds.), Vol. 43. 14–28. http://dblp.uni-trier.de/db/series/eptcs/eptcs43.html#abs-1012-4896

Andreas Abel. 2012. Type-Based Termination, Inflationary Fixed-Points, and Mixed Inductive-Coinductive Types. In *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012, Tallinn, Estonia, 24th March 2012.* 1–11. https://doi.org/10.4204/EPTCS.77.1

Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. 2010. ΠΣ: Dependent Types Without the Sugar. In *Proceedings of the 10th International Conference on Functional and Logic Programming (FLOPS'10).* 40–55.

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of Path-dependent Types. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14).* ACM, New York, NY, USA, 233–249. https://doi.org/10.1145/2660193.2660216

Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. (2018). www.irif.fr/~sozeau/research/publications/drafts/Towards_Certified_Meta-Programming_with_Typed_Template-Coq.pdf

Leif Andersen, Stephen Chang, and Matthias Felleisen. 2017. Super 8 Languages for Making Movies (Functional Pearl). *Proc. ACM Program. Lang.* 1, ICFP, Article 30 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110274

Lennart Augustsson. 2007. Simpler, Easier! http://augustss.blogspot.ru/2007/10/simpler-easier-in-recent-paper-simply.html

Andrej Bauer. 2012. How to Implement Dependent Type Theory. http://math.andrej.com/2012/11/08/how-to-implement-dependent-type-theory-i/

Jasmin C. Blanchette, Cezary Kaliszyk, Lawrence C. Paulson, and Josef Urban. 2016. Hammering towards QED. *J. Formaliz. Reason.* 9, 1 (2016), 101–148.

Edwin Brady and Kevin Hammond. 2006. Dependently Typed MetaProgramming. In *7th Symposium on Trends in Functional Programming.*

Edwin C. Brady. 2005. *Practical Implementation of a Dependently Typed Functional Programming Language.* Ph.D. Dissertation. University of Durham.

Stephen Chang, Alex Knauth, and Ben Greenman. 2017. Type Systems As Macros. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages.* 694–705.

David Christiansen and Edwin Brady. 2016. Elaborator Reflection: Extending Idris in Idris. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016).* ACM, New York, NY, USA, 284–297. https://doi.org/10.1145/2951913.2951932

Jesper Cockx, Dominique Devriese, and Frank Piessens. 2014. Pattern Matching Without K. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14).* ACM, New York, NY, USA, 257–268. https://doi.org/10.1145/2628136.2628139

Thierry Coquand. 1992. Pattern Matching with Dependent Types. In *Proceedings of the Workshop on Types for Proofs and Programs.* 71–83.

Thierry Coquand and Gérard P. Huet. 1988. The Calculus of Constructions. *Inf. Comput.* 76, 2/3 (1988), 95–120. https://doi.org/10.1016/0890-5401(88)90005-3

Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. 2006. A Core Calculus for Scala Type Checking. In *Proceedings of the 31st International Conference on Mathematical Foundations of Computer Science (MFCS'06).* Springer-Verlag, Berlin, Heidelberg, 1–23. https://doi.org/10.1007/11821069_1

N.G. de Bruijn. 1991. Telescopic Mappings in Typed Lambda-Calculus. *Information and Computation* 91, 2 (1991), 189–204.

David Delahaye. 2000. A Tactic Language for the System Coq. In *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning (LPAR'00).* Springer-Verlag, Berlin, Heidelberg, 85–95. http://dl.acm.org/citation.cfm?id=1765236.1765246

Dominique Devriese and Frank Piessens. 2013. Typed Syntactic Meta-programming. In *of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP 2013).* 73–86.

Peter Dybjer. 1994. Inductive families. *Formal Aspects of Computing* 6, 4 (01 Jul 1994), 440–465.

R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. 1992. Syntactic Abstraction in Scheme. *Lisp Symb. Comput.* 5, 4 (Dec. 1992), 295–326. https://doi.org/10.1007/BF01806308

Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A metaprogramming framework for formal verification. *Proceedings of the ACM on Programming Languages (PACMPL)* 1, ICFP (2017), 34:1–34:29. https://doi.org/10.1145/3110278

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2015. The Racket Manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015).* 113–128.

Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. 2018. A Programmable Programming Language. *Commun. ACM* 61, 3 (Feb. 2018), 62–71. https://doi.org/10.1145/3127323

Matthew Flatt. 2002. Composable and Compilable Macros: You Want It when?. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02)*. ACM, New York, NY, USA, 72–83. https://doi.org/10.1145/581478.581486

Matthew Flatt. 2016. Binding As Sets of Scopes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 705–717.

Matthew Flatt, Ryan Culpepper, David Darais, and Robert Bruce Findler. 2012. Macros That Work Together: Compile-time Bindings, Partial Expansion, and Definition Contexts. 22, 2 (March 2012), 181–216. https://doi.org/10.1017/S0956796812000093

Eduardo Giménez. 1995. Codifying Guarded Definitions with Recursive Schemes. In *International Workshop on Types for Proofs and Programs (TYPES)*. https://doi.org/10.1007/3-540-60579-7_3

Healfdene Goguen, Conor McBride, and James McKinna. 2006. Eliminating Dependent Pattern Matching. In *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 521–540. https://doi.org/10.1007/11780274_27

Georges Gonthier and Assia Mahboubi. 2010. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning* 3, 2 (2010), 95–152. https://hal.inria.fr/inria-00515548

Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. 2011. How to Make Ad Hoc Proof Automation Less Ad Hoc. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 163–175. https://doi.org/10.1145/2034773.2034798

John Hughes, Lars Pareto, and Amr Sabry. 1996. Proving the Correctness of Reactive Systems Using Sized Types. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '96)*. ACM, New York, NY, USA, 410–423. https://doi.org/10.1145/237721.240882

Joomy Korkut and David Christiansen. 2018. Extensible Type-Directed Editing. In *Proceedings of the Workshop on Type-Driven Development*.

Robbert Krebbers, Amin Timany, and Lars Birkedal. 2017. Interactive Proofs in Higher-order Concurrent Separation Logic. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 205–217. https://doi.org/10.1145/3009837.3009855

Andres Löh, Conor McBride, and Wouter Swierstra. 2010. A Tutorial Implementation of a Dependently Typed Lambda Calculus. *Fundam. Inform.* 102, 2 (2010), 177–207.

Gregory Malecha and Jesper Bengtson. 2016. *Programming Languages and Systems: 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Springer Berlin Heidelberg, Berlin, Heidelberg, Chapter Extensible and Efficient Automation Through Reflective Tactics, 532–559. https://doi.org/10.1007/978-3-662-49498-1_21

Alberto Martelli and Ugo Montanari. 1982. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.* 4, 2 (April 1982), 258–282. https://doi.org/10.1145/357162.357169

Per Martin-Löf. 1975. An intuitionistic theory of types: Predicative part. *Studies in Logic and the Foundations of Mathematics* 80 (1975), 73–118.

Conor McBride. 2000. *Dependently Typed Functional Programs and Their Proofs*. Ph.D. Dissertation. University of Edinburgh, UK. http://hdl.handle.net/1842/374

Bengt Nordström, Kent Petersson, and Jan M. Smith. 1990. *Programming in Martin-Löf's Type Theory: An Introduction*. Clarendon Press, New York, NY, USA.

Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph.D. Dissertation. Chalmers University of Technology. http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf

Brigitte Pientka. 2008. A Type-theoretic Foundation for Programming with Higher-order Abstract Syntax and First-class Substitutions. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. 371–382.

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2018. *Logical Foundations*. Electronic textbook.

Benjamin C. Pierce and David N. Turner. 1998. Local Type Inference. In *Proceedings of the 25th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*. 252–265.

Justin Pombrio and Shriram Krishnamurthi. 2015. Hygienic Resugaring of Compositional Desugaring. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 75–87. https://doi.org/10.1145/2784731.2784755

Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. 2007. Ott: Effective Tool Support for the Working Semanticist. In *of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP 2007)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/1291151.1291155

Antonis Stampoulis and Zhong Shao. 2010. VeriML: Typed Computation of Logical Terms Inside a Language with Effects. In *of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP 2010)*. 333–344.

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 530–541.

Stephanie Weirich. 2014. Pi Forall: notes from OPLSS. https://github.com/sweirich/pi-forall

Stephanie Weirich, Antoine Voizard, Pedro Henrique Avezedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proceedings of the ACM on Programming Languages (PACMPL)* 1, ICFP (Aug. 2017). https://doi.org/10.1145/3110275

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

Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon L. Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. 2012. Giving Haskell a promotion. In *Types in Language Design and Implementation (TLDI)*. https://doi.org/10.1145/2103786.2103795

Beta Ziliani, Derek Dreyer, Neelakantan R Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. 2013. Mtac: A Monad for Typed Tactic Programming in Coq. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP 2013)*. ACM, New York, NY, USA, 87–100. https://doi.org/10.1145/2500365.2500579

Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Conference on Computer and Communications Security, (CCS)*. https://doi.org/10.1145/3133956.3134043