Dependent Type Systems as Macros

STEPHEN CHANG, Northeastern University, USA
MICHAEL BALLANTYNE, Northeastern University, USA
MARCELA POFFALD, Unaffiliated, USA
WILLIAM J. BOWMAN, Northeastern University, USA

Increasingly, programmers want the power of dependent types, yet significant expertise is still required to write realistic dependently-typed programs. In response, domain-specific languages (DSLs) attempting to tame dependent types have proliferated, adding notation and tools tailored to a problem domain. This only shifts the problem, however, since implementing such languages requires at least as much expertise as using them.

We show how to lower the burden for implementing dependently-typed languages and DSLs, using a classic approach to DSL implementation not typically associated with typed languages: macros. By leveraging a macro system, programmers may reuse all of a host language's infrastructure when implementing a new, dependently-typed language or DSL, reducing the overall effort. We also extend the Turnstile language, a meta-DSL for implementing typed DSLs using syntax resembling "pen and paper" models, with support for dependent types. Using macros simplifies not only the initial language implementation, but also the addition of extensions like notation or tactic languages—all but required features for dependently-typed languages.

We evaluate our approach by building three languages in different parts of the design space: first, we present a video-editing DSL with a Dependent ML-like type system, demonstrating that our approach accommodates "lightweight" dependent types; second, we gradually extend MLTT to the Calculus of Inductive Constructions, demonstrating that our approach is modular, and scales to "heavyweight" dependent type systems; finally, we describe Cur, a prototype proof assistant with a design similar to Coq, which supports new notation and an extensible tactic language, demonstrating that our approach scales to realistic dependently-typed languages.

1 INTRODUCTION

Programmers are increasingly wanting and using dependent types. For example, Haskell has embraced type-level computation [Weirich et al. 2017], Rust is considering adding Π types [rus 2017], and new dependently typed languages such as F* have leveraged domain-specific languages (DSLs) to verify software such as Firefox's TLS [Beurdouche et al. 2017; Zinzindohoué et al. 2017].

Despite this progress, implementing and using dependent types remains complicated, and thus not all programmers are ready for them. At one end of the spectrum, language designers debate about the "right" amount of dependent types. For example, determining the ideal "power-to-weight" ratio has slowed adoption in Haskell [Yorgey et al. 2012] and has led to repeated rewrites of Rust's dependent type RFCs [rus 2016]. At the other end, proof assistants that ignore "weight" in favor of "power" must layer on companion DSLs (*e.g.*, a tactic language) to help programmers use the language [Brady and Hammond 2006; Christiansen 2014; Devriese and Piessens 2013; Ebner et al. 2017; Gonthier and Mahboubi 2010; Gonthier et al. 2011; Krebbers et al. 2017; Malecha and Bengtson 2016; Pientka 2008; Stampoulis and Shao 2010; Ziliani et al. 2013].

Ideally, language designers or even users would simply construct a new DSL for each problem domain, choosing how much "power" to wield on a case-by-case basis. Indeed, DSLs have been used effectively to tame dependent types [Barthe et al. 2009; Chlipala 2011; Chlipala et al. 2017; Zinzindohoué et al. 2017], but so far, they have not been simple to build.

Authors' addresses: Stephen Chang, Northeastern University, USA, stchang@ccs.neu.edu; Michael Ballantyne, Northeastern University, USA, mballantyne@ccs.neu.edu; Marcela Poffald, Unaffiliated, USA, mpoffald@gmail.com; William J. Bowman, Northeastern University, USA, wjb@williamjbowman.com.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

We aim to change that, by showing how to use *macros* to create dependently-typed DSLs. Procedural macros, in the style of LISP and its descendants, simplify the construction of DSLs [Fowler and Parsons 2010] by reusing much of the host language infrastructure such as parsing, elaboration, namespace management, and compilation. We show how this approach reduces the complexity of dependent types for both language implementors—because the DSL may reuse the infrastructure of the host language—and users—because the complexity of the type system may be exactly tailored to a specific problem domain. Better yet, any languages created with this approach may leverage the macro system to implement extensions to the core language such as new notation and tools for automatically constructing proofs and programs.

The macro-based approach to building DSLs has not historically included typed languages, but recently Chang et al. [2017] introduced the technique of "type systems as macros", showing how programmers may use macros to create typed DSLs as well. Specifically, they show that with a contemporary macro system as found in Racket [Flatt and PLT 2010]—a LISP and Scheme descendant—programmers may create typed DSLs simply by embedding type rule logic directly into the macro definitions. This macro-based approach improves on the traditional approach of creating typed DSLs—where domain-specific types are encoded into an existing host type system—because it does not constrain DSL creators to the limitations of any particular type system. Instead, DSL implementers have the flexibility to create the right type system for their domain. Finally, macros are naturally expressed as local, modular transformations, and implementing type rules with macros results in a type checker that naturally matches the modularity of its mathematical specification. This is demonstrated in Chang et al. [2017]'s TURNSTILE, a meta-DSL that allows implementing typed DSLs using a judgement-like syntax resembling what programmers would find in a textbook.

We extend the "type systems as macros" approach—and the TURNSTILE language—to support creating dependently-typed languages. This is a major technical challenge, as dependent types break many of the assumptions implicit in macro systems, and in the previous design of TURNSTILE. For example, run-time and expansion time are distinct phases for a macro system, but there can be no such distinction for a dependently-typed language, which may evaluate expressions while type checking. There are new design challenges as well, *e.g.*, a framework for building dependently-typed languages should support common type notation such as *telescopes*, *i.e.*, nested binding environments [de Bruijn 1991; McBride 2000]. Specifically, we make the following contributions.

- We extend TURNSTILE with a new API for defining types that is syntactically concise, yet robust enough to implement a range of constructs from base types, to binding forms like Π types, to indexed inductive type families.
- We show how to leverage macros and macro expansion to perform the work of type-level reduction in an extensible manner, and also add TURNSTILE constructs that allow implementing these reduction rules with familiar on-paper syntax.
- A key source of complexity in implementing dependent types is handling dependent binding structure, *e.g.*, manipulating *telescopes*. For example, checking such binding types requires interleaving checking with adding new environment bindings, and instantiating them requires a folding substitution operation. We extend TURNSTILE's pattern language to support these operations, allowing us to express features with complex binding structure (such as indexed inductive type families) using a concise, intuitive notation.
- We evaluate our approach by constructing three example languages.

(1) We present a video-editing DSL with a Dependent ML-like type system that statically enforces guarantees about the lengths of videos, tracks, and playlists, demonstrating that our approach allows tailoring dependent types into a more "lightweight" flavor.

(2) We gradually build up a core calculus, culminating in the Calculus of Inductive Constructions (CIC) [Pfenning and Paulin-Mohring 1989], demonstrating that our macros-based approach is modular, extensible, and supports "heavyweight" dependent type systems.
(3) To demonstrate that our approach supports creating realistic languages, we present Cur, a prototype proof assistant whose core—the impredicative CIC—resembles that of Coq, but requires only a few dozen lines of code in our extended Turnstile. With macros, we easily extend core Cur with features such as syntactic sugar and a tactic language. Using the latter, we worked through several chapters of "Logical Foundations" [Pierce et al. 2018], demonstrating that the tactic system is sophisticated enough to support Coq-style proofs.

2 CREATING MACRO-BASED DSLS WITH RACKET: PRIMER

This section introduces building languages—typed and untyped—with Racket's macro system.[1]

```
#lang racket                                                      bool-lang
(provide true false and or not #%app ⊃ (rename truth-table λ))

(define-m (truth-table (x^id ...) [arg^bool ... = res^bool] ...)
  #:with (dnf-clause-fn ...) (λ (x ...) (and res ((bool->lit arg) x) ...))
  (λ (x ...) (or (dnf-clause-fn x ...) ...)))

(define ⊃ (truth-table (x y) [false false = true]
                             [true  false = false]
                             [false true  = true]
                             [true  true  = true]))

(define-m bool->lit [(_ true) (λ (x) x)] [(_ false) not])
```

Fig. 1. A basic (untyped) Boolean-logic DSL created with Racket.

2.1 An Untyped DSL

A Racket language is defined by the exports of a module. Figure 1 presents the bool-lang module, an example language of Boolean logic, which we use to introduce notation used in the paper,[2] and DSL creation with macros. Key to defining languages as macros are the abilities to:

(1) *reuse* host language (Racket) features for their own language; *e.g.*, bool-lang reuses `true`, `false`, `and`, `or`, `not`, as well as the function application form `#%app`;
(2) *add* functions and forms; *e.g.*, bool-lang defines and exports the implication function ⊃;
(3) *interpose* on primitive forms, such as functions and application, using syntactic hooks such as `#%app` and λ, *e.g.*, bool-lang redefines λ by the `truth-table` macro;
(4) *exclude* features from the host language; *e.g.*, bool-lang does not include first-class functions, numbers, or lists, but only the explicitly exported features.

To program with bool-lang, programmers use the `#lang` directive:

---

[1] Appendix B lists the macro system features we use in more detail, and discusses other languages with the same features.

[2] To more clearly communicate concepts, we sacrifice code precision by stylizing code with abbreviations or non-syntactic elements like color and subscripts. For example, `define-m` is shorthand for a Racket macro defined via `define-syntax` and the `syntax-parse` pattern matching construct [Culpepper and Felleisen 2010]. Examples may not run as presented, but full implementations of all examples are available at https://www.github.com/stchang/macrotypes, https://www.github.com/wilbowma/cur. We summarize our style conventions in Appendix A.

bool-prog

```
#lang bool-lang
(⊃ true false)  ; result: false
((λ (x) [true = false] [false = true]) (⊃ false true))  ; result: false
(+ 1 2)  ; ERR: unsupported
```

In bool-lang's implementation, `truth-table` is a macro, *i.e.*, it is defined[3] with `define-m`, that converts a table of boolean values into a function implementing an equivalent formula in disjunctive normal form (DNF). Macros consume and produce a *syntax object*, an AST data structure that combines a tree of symbols with context information like source location and binding structure. A macro typically pattern-matches on its input, using a *syntax pattern* (`green` in this paper) whose shape dictates how the macro is invoked. (Note that the name of the macro being invoked is included as part of the input for the macro, so all initial syntax patterns begin with a pattern representing the macro's own name. For example, `truth-table` is part of the syntax pattern in the definition of `truth-table`.) Most identifiers in a syntax pattern are bound as *pattern variables*, which are associated with corresponding pieces of syntax supplied by the programmer when they invoke the macro. The ellipses pattern ... means "zero of more of the preceding pattern".

bool-lang invokes `truth-table` to define ⊃, where the pattern `(x ...)` in `truth-table`'s input pattern matches syntax object `(x y)`, representing the arguments expected by ⊃. A superscript *syntax class* may adorn a pattern variable, which refines what syntax matches that pattern variable. For example `truth-table`'s input parameters are tagged with `id`, so they only match identifiers. Further, its body consists of rows of literal boolean values representing the inputs and output, separated by **=** (a bolded pattern symbol, e.g., **=**, denotes an exact value that must be matched). A `#:with` keyword introduces additional pattern variables by matching on the syntax object computed by the second position after the keyword. For example, `truth-table` uses `#:with` to define `dnf-clause-fn` pattern variables, representing the "and" clauses in its "or of ands" DNF output.

Pattern variables are used in *syntax templates* (`blue` in the this paper). A template replaces references to pattern variables with their corresponding syntax object values. Ellipses that followed a variable in a syntax pattern must also accompany references to that variable in the syntax template. Macros frequently use syntax templates to construct their outputs, *e.g.*, `truth-table`'s output is a template that references the `dnf-clause-fn` pattern variables.

Finally, the meaning of any non-pattern-variable identifiers in a syntax template is taken from the context of the macro definition. For example, the syntax template constructing `dnf-clause-fn` references Racket's λ and `and`, as well as a local macro `bool->lit`, which converts a boolean value into DNF formula literal. The `bool->lit` macro uses an alternate macro definition syntax with multiple clauses, whose patterns are tried in order. Observe that each pattern still includes the name of the macro in its first position, but our example ignores it using the _ pattern.

2.2 A Typed DSL

Figure 2 (left) presents typed-lang, which adds arithmetic to bool-lang, and a type system to ensure that operations receive the right values. It is created with the "type systems as macros" technique [Chang et al. 2017] and uses the same four "DSL tools" from Section 2.1. Specifically, typed-lang interposes on λ and `#%app` with two new macros, `typed-λ` and `typed-app`, respectively, replacing the `#%app` and (truth table) λ from bool-lang. These macros use the computation and pattern matching performed by `#:with` to implement basic type checking. Since macros embody local transformations, however, successfully checking types in this manner requires additional coordination between macros, to communicate type information. Solving this coordination problem

---

[3] We underline names being defined.

```
#lang racket                               typed-lang
(provide [typed-λ λ] [typed-app #%app]
         [typed+ +] [typed-and and])
(define-m (typed-app f e)
 #:with [f̅ (→ τ_in τ_out)] (synth f)
 #:with e̅ (check e τ_in)
 (assign (#%app f̅ e̅) τ_out))


(define-m (typed-λ [x^id : τ_in] e)
 #:with [x̅ e̅ τ_out] (synth e #:ctx [x : τ_in])
 (assign (λ x̅ e̅) (→ τ_in τ_out)))


(define-m (typed-and e1 e2)
  #:with e̅1 (check e1 Bool)
  #:with e̅2 (check e2 Bool)
  (assign (and e̅1 e̅2) Bool))


(define-m typed+ (assign + (→ Int Int Int)))
```

```
#lang turnstile                            typed-lang
(provide [typed-λ λ] [typed-app #%app]
         [typed+ +] [typed-and and])
(define-tyrule (typed-app f e) ≫
 [⊢ f ≫ f̅ ⇒ (→ τ_in τ_out)]
 [⊢ e ≫ e̅ ⇐ τ_in]
 -------------------------------
 [⊢ (#%app f̅ e̅) ⇒ τ_out])

(define-tyrule (typed-λ [x^id : τ_in] e) ≫
  [[x ≫ x̅ : τ_in] ⊢ e ≫ e̅ ⇒ τ_out]
  --------------------------------
  [⊢ (λ x̅ e̅) ⇒ (→ τ_in τ_out)])

(define-tyrule (typed-and e1 e2) ≫
 [⊢ e1 ≫ e̅1 ⇐ Bool]
 [⊢ e2 ≫ e̅2 ⇐ Bool]
 ---------------------
 [⊢ (and e̅1 e̅2) ⇒ Bool])

(define-primop typed+ + : (→ Int Int Int))
```

Fig. 2. (Part of) a typed extension of BOOL-LANG, (left) using Racket, and (right) using TURNSTILE.

is the essence of "type systems as macros". Specifically, the `synth`, `check`, and `assign` metafunctions (Figure 3) implement such a communication protocol between type-checking macros.

The `typed-app` macro first uses `synth` to compute the type of function term `f`, which must match the pattern `(→ τin τout)`. The `synth` function produces a second result `f̅` representing an elaborated version of `f`. Because our type checker is embedded in macro definitions, type checking is interleaved with macro expansion, and `synth` necessarily expands `f`. To avoid redundant expansions, `synth` returns the elaborated `f̅` so that `typed-app` may produce an elaborated term that includes `f̅`.[4] This turns out to be an effective and concise way to implement type checkers, since type systems often require an elaboration pass anyway, *e.g.*, for type erasure.

The `typed-app` macro's second premise uses `check` to ensure that argument `e` has type $\tau_{\mathtt{in}}$. Similar to `synth`, `check` expands its argument and returns the elaborated `e̅`. Finally, `typed-app` constructs output term `(#%app f̅ e̅)`, which uses the (untyped) host language `#%app`, and "assigns" it type $\tau_{\mathtt{out}}$. This call to `assign` is the crucial step that communicates type information between macros, by attaching type information to the syntax objects, which other type checking macros understand.

In `typed-λ`, `synth` computes the type of body `e` in a context where `x` has type $\tau_{\mathtt{in}}$. (The function is passed the type environment via a named keyword argument `#:ctx`.) Here `synth` returns the elaborated `e̅`, as well as the binder `x̅` for references in `e̅`. The latter is required to construct the output term in a hygienic macro system, *i.e.*, one that tracks and enforces proper binding structure in all syntax objects. In other words, programmers may not create binding terms using any arbitrary identifier with the same name; instead a proper binder must carry the correct program context information, added via expansion (see Flatt [2016] for more details). Thus only `x̅` may close over `e̅` because they were expanded with the same context. It turns out that this knowledge of the program's binding structure is extremely useful for implementing type systems and many type

---

[4] We use overlines to denote pattern variables bound to fully elaborated syntax

operations such as substitution and alpha equivalence. Finally, `typed-λ` uses `assign` to associate the elaborated syntax `(λ x̄ ē)` with its type (→ $\tau_{\mathtt{in}}$ $\tau_{\mathtt{out}}$).

Figure 2 (left) includes a few other "type rules": `typed-and` checks that its arguments are `Bool`, and `typed+` is a function with type `(→ Int Int Int)`. In the latter case, `typed+` is an *identifier macro* that does not require any arguments to invoke it. Thus, `typed-app` handles type checking application of `typed+`. For now, we assume that types are literal pieces of syntax as in Figure 2 (left), *e.g.*, `Bool` and `Int`. Section 3.1 presents a more thorough treatment of defining types.

```
(define (assign e τ) (attach e 'type τ))        (define (check e τ #:ctx [ctx ()])
(define (synth e #:ctx [ctx ()])                  #:with [x̅s̅ ē τ_e] (synth e τ #:ctx ctx)
  #:with [x̅s̅ ē] (local-expand (let_stx ctx e))   (if (τ= τ_e τ) (x̅s̅ ē)
  (x̅s̅ ē (detach ē 'type)))                                     (err "type mismatch")))
```

Fig. 3. "Type systems as macros" core API.

Figure 3 shows the implementations of `synth`, `check`, and `assign`. They require only a few lower-level operations on syntax, demonstrating that the entire type checker is implemented "as macros". Specifically, the functions rely on two features: (1) a `local-expand` function that initiates macro expansion on a syntax object, which allows invoking "type checking" macros on a subterm; and (2) a way of associating additional information (types) with syntax objects; we use *syntax properties* which, via `attach` and `detach`, to associate key-value pairs to syntax objects.

Individually, `assign` attaches a type to a term, at key *'type*. The `synth` function consumes an expression `e` and an optional environment `ctx`—which has shape `([x : τ] ...)`—and invokes the macro expander via `local-expand` to type check `e`. To implement the type environment, it wraps `e` with `let`$_{\mathtt{stx}}$, which allows defining local macros. In other words, `let`$_{\mathtt{stx}}$ defines new typed macros such that untyped variable references in `ē` are themselves macro invocations that return the desired type information, effectively using the macro environment to implement the type environment. Finally, `synth` returns a triple consisting of the expanded context variables, the expanded term `ē`, and its type. The coloring of `synth`'s output denotes a *quasiquoted* syntax template, *i.e.*, the syntax object is constructed with references to pattern variables `x̅s̅` and `ē`, and a call to the (meta)function `detach`. 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 paper, we assume τ= (not shown) is syntactic equality up to alpha-equivalence; it is straightforward to implement since syntax objects are aware of the program's binding structure.

2.3 A DSL for Typed DSLs

Chang et al. [2017] observed that Figure 2 (left)'s macros closely correspond to algorithmic specifications. Thus, they created Turnstile, a meta-DSL that allows writing type-checking macros using a type judgement-like syntax, as seen in Figure 2 (right). Specifically, Turnstile uses two relations which correspond to "synth" and "check" bidirectional type checking judgments [Pierce and Turner 1998], interleaved with elaboration. They are implemented with `synth` and `check` from Figure 3, respectively. The judgement `[ctx ⊢ e ≫ ē ⇒ τ]` says that, in environment `ctx`, `e` elaborates (≫) to `ē` and synthesizes (⇒) type τ—*i.e.*, τ is an output. Observe how the syntax pattern and syntax templates on the left and right of Figure 2 remain the same. The check judgment `[ctx ⊢ e ≫ ē ⇐ τ]` specifies that, in environment `ctx`, `e` elaborates to `ē` and checks against (⇐) type τ—*i.e.*, τ is an input. Bindings are added to the type environment by writing them to the left of ⊢, as in `typed-λ`, but only *new* variables must be written. Since Turnstile reuses the macro environment as the type environment, existing bindings are automatically propagated by lexical scope. Figure 2 (right) uses `define-tyrule`, which has a few usage variations, "synth" (L) and "check" (R):

```
(define-tyrule input-pattern ≫
  premises ...
  --------------
  [⊢ output-template ⇒ type])
```

```
(define-tyrule input-pattern ⇐ input-type ≫
  premises ...
  --------------
  [⊢ output-template])
```

Figure 2 (right) implements "synth" rules, which fire when a term matches `input-pattern`. If the `premises`—a series of synth and check judgements—holds, the macro produces the output specified by `output-template`, with `type` attached. Observe that a textbook would typically write these rules with the entire conclusion [⊢ `input-pattern` ≫ `output-template` ⇒ `type`] at the bottom, but Turnstile shifts the conclusion's input (pattern) to the top, as in a typical macro definition. With a "check" rule, `input-type` is also an input and is written next to `input-pattern`, above the `premises`. Thus, a "check" type rule fires when: a term matches `input-pattern`, the term's type may be inferred from its context, *and* that type matches pattern `input-type`. Turnstile automatically switches from "check" to "synth" rules when no corresponding "check" rules exists.

3 LIGHTWEIGHT DEPENDENT TYPES, FOR VIDEO

While Chang et al. [2017] implement a variety of languages with Turnstile, they can not handle dependent types since they assume an explicit phase distinction, *i.e.*, that terms and types are distinct. We show that maintaining this distinction is no longer possible when implementing dependent types, and how to improve Turnstile to cover this deficit. We do this in the context of an example, 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] do briefly describe a few type rules, our work is the first to explain the underlying details required to implement indexed types as macros, such as the implementation of types and type-level computation. We focus on the new techniques required to express indexed types as macros, using Typed Video as an example, not on the use or implementation of Typed Video itself.

Typed Video is a typed version of Andersen et al. [2017]'s video language, a DSL for editing movies that has been used to create the video proceedings of several workshops, *e.g.*, OPLSS.[5] Typed Video uses indexed types in order to statically rule out errors that arise when creating and combining video streams. 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 producers ultimately represent physical data on disk, it's possible to crash a program (usually during rendering) by accidentally using more data than exists. To prevent this, Typed Video assigns producer values a `Producer` type, indexed by its length. This is an ideal type system for Video since programmers are already required to provide the length of many expressions.

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

video-prog

```
#lang typed/video
(define (mk-conf-talk [n : Int] [aud : (Producer n)] [vid : (Producer n)]
                      [slid : (Producer n)]) #:when (> n 3) -> (Producer (+ n 9))
  (playlist (img "conf-logo.png" #:len 9)
            (fade #:len 3)
            (overlay aud vid sli)))
```

The `mk-conf-talk` function consumes an integer length `n` and audio, video, and slides producers with types `(Producer 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

---

[5] https://lang.video/community.html

an 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 9 frames, to account for the added logo. The function is assigned the type:

```
(→vid [n : Int] [aud : (Producer n)] [vid : (Producer n)] [sli : (Producer n)]
      (Prod (+ n 9)) #:when (> n 3))
```

In this binding variant of a function type, each argument is named and the type of each argument may reference the names preceding it. We demonstrate how to scale the "type systems as macros" approach to support this binding structure, and extend the Turnstile implementation with the *telescope* notation used on-paper to manipulate types like this [de Bruijn 1991; McBride 2000].

3.1 Defining Types

The first challenge is how one can define a typing rule for the function type just presented. This subsection describes how to extend the key typing judgments when encoded in macros (introduced in Section 2), and demonstrates our implementation of this approach via a new Turnstile API for defining types, including dependent types.

To type check *types themselves*, the obvious approach is to define *types as macros*, not just terms. We can then specify the semantics for type construction in the same judgments we used in Section 2. Figure 4 shows type rules for (single-arity) function types → and $\rightarrow_{\texttt{vid}}$.

```
(struct →̅ (in out))
(define-tyrule (→ τin τout) ≫
  [⊢ τin ≫ τ̅i̅n̅ ⇐ Type]
  [⊢ τout ≫ τ̅o̅u̅t̅ ⇐ Type]
  ----------
  [⊢ (#%app →̅ τ̅i̅n̅ τ̅o̅u̅t̅) ⇒ Type])
```

```
(struct →̅v̅i̅d̅ (in out))
(define-tyrule (→vid [x : τin] τout) ≫
  [⊢ τin ≫ τ̅in ⇐ Type]
  [[x ≫ x̅ : τ̅in] ⊢ τout ≫ τ̅out ⇐ Type]
  -----------------------------
  [⊢ (#%app →̅v̅i̅d̅ τ̅in (λ (x̅) τ̅out)) ⇒ Type])
```

Fig. 4. Some type rules (macro definitions) for single-arity function types, (left) →, (right) $\rightarrow_{\texttt{vid}}$

In Figure 4 (left), the rule for a standard function type → checks that its input and output have type `Type`, a type of types.[6] But what should be the output of the rule? In other words, what is the "runtime" representation of a type? For typed lambda, we used the underlying host language's lambda representation, but there is no analogous construct for types. Thus, for types, we have more freedom in what to put in the rule's output. Any representation, however, has three criteria. The first two, which are simple to understand, are: 1) it should uniquely identify the type and 2) it should store the arguments to the type constructor. In Figure 4 (left), we use a named record $\overline{\rightarrow}$, declared with `struct`, to represent the → type. Thus, the output of the → macro is a *syntax object* of an application of $\overline{\rightarrow}$ to its arguments.

The third criteria requires thinking about binding. In Figure 4 (right), the rule for $\rightarrow_{\texttt{vid}}$ resembles that of →, but differs in that: a), the type's input has a name `x`; b) the output $\tau_{\texttt{out}}$ is checked in the context of `x` because it may reference `x`; and c) for the type's representation in the rule's conclusion, a *lambda wraps and binds* references to $\overline{\texttt{x}}$ in $\overline{\tau}_{\texttt{out}}$. The last difference reveals the third criteria for a type's internal representation: it must comply with hygiene. Syntax objects must have a valid binding structure at all times, or they are rejected during macro expansion. This last criteria is key to getting boilerplate operations—such as substitution, alpha-equivalence, and environment management—for free.

---

[6] This paper omits discussing the implementation of `Type`, which is not interesting, due to space. Note that the Cur language in Section 5 supports a proper type universe hierarchy, as found in languages like Coq.

3.2 Type Checking Telescopes

The above tells us how to represent dependent types as macros, but dependent types and their binding structure also complicate defining macros by pattern matching. Suppose we want to change our function type rules to accommodate multiple arguments. In Figure 5 (left), the plain function type may simply use the ellipses pattern, which effectively "maps" over the τ_{in}s. For dependent types, as in for $\rightarrow_{\text{vid}}$, this "map" operation is incorrect, and results in the wrong binding structure. For example, the $\rightarrow_{\text{wrong}}$ rule in Figure 5 (right) tries to use the same ellipses pattern as on the left, but this type checks each argument's type τ_{in} in a type environment with *every* argument `x` bound, *including itself*. On the left, without dependent types, this is not a problem since types cannot reference the term variables from the same type annotation.

```
(struct →̅ (in out))
(define-tyrule (→ τ_in ... τ_out) ≫
  [⊢ τ_in ≫ τ̅_in ⇐ Type] ...
  [⊢ τ_out ≫ τ̅_out ⇐ Type]
  ----------
 [⊢ (#%app →̅ (τ̅_in ...) τ̅_out) ⇒ Type])
```

```
(struct →̅_vid (types))
(define-tyrule (→_wrong [x : τ_in] ... τ_out) ≫
 [[x ≫ x̅ : τ̅_in] ... ⊢ [τ_in ≫ τ̅_in ⇐ Type] ...]
 [[x ≫ x̅ : τ̅_in] ... ⊢ τ_out ≫ τ̅_out ⇐ Type]
 ----------------------------
 [⊢ (#%app →̅_vid (λ (x̅ ...) τ̅_in ... τ̅_out)) ⇒ Type])

(define-tyrule (→_vid [x : τ_in] ... τ_out)
 [[x ≫ x̅ : τ_in ≫ τ̅_in ⇐ Type] ... ⊢ τ_out ≫ τ̅_out ⇐ Type]
  ----------------------------
  [⊢ (#%app →̅_vid (λ (x̅ ...) τ̅_in ... τ̅_out)) ⇒ Type])
```

Fig. 5. Some type rules for multi-arity function types, (left) →, (right) $\rightarrow_{\text{vid}}$

Instead, to use familiar macro notation to implement dependent types, we require that ellipses express a "fold" operation for recursively applying macro expansion (*i.e.*, type checking) to express the proper binding structure. Since environments themselves contain type annotations, this fold operation must interleave binding and checking. In Figure 6, we present such a fold operation which is part of the "dependent type systems as macros" core API (*i.e.*, an extension to the core API discussed in Section 2 needed to support dependent types). This new function consumes a name `x`, a target to check τ, an expected type κ_{expected} for τ, and a previous context, and it checks that τ has type κ_{expected} *while* adding `x` and τ to create the next context. This new context is returned along with expanded versions of `x` and τ.

```
(define (folding-check x τ κ_expected #:ctx [ctx_prev ()])
  #:with [ctx_new x̅ τ̅] (synth x #:ctx ([x τ] ctx_prev))
  #:with κ (detach τ̅ 'type)
  (if (τ= κ κ_expected) (ctx_new x̅ τ̅) (err "type mismatch")))
```

Fig. 6. A folding variant of the check API function from Figure 3.

In our extension to TURNSTILE, we interpose on the ellipses to use `folding-check` instead of `check` when appropriate. In Figure 5 (right), we give the corrected definition of $\rightarrow_{\text{vid}}$ using the new TURNSTILE syntax `[x ≫ x̅ : τ_in ≫ τ̅_in ⇐ Type] ...`, which checks each τ_{in}, but also names it so that subsequent type checking invoked by the ellipses may reference the argument. Since the new syntax both checks and binds, subsuming what is typically on the left *and* right side of the (⊢), programmers may use it on *either side*, *e.g.*, the following is equivalent to the definition in Figure 5:

```
(define-tyrule (→_vid [x : τ_in] ... τ_out)
  [⊢ [x ≫ x̅ : τ_in ≫ τ̅_in ⇐ Type] ... [τ_out ≫ τ̅_out ⇐ Type]]
  ----------------------------
```

```
  [⊢ (#%app →̅vid (λ (x̅ ...) τ̅in ... τ̅out)) ⇒ Type])
```

Note that a single lambda wrapping all the types in the macro's output is sufficient due to hygiene. There will be no capture so long as each type was expanded in the appropriate context.

3.3 Macros for Pattern Matching

In general, the programmer does not need to know the underlying "run-time" type representation, and would prefer to simply pattern match on the *surface* syntax of the type instead of the "run-time" representation produced by macro expansion. In TURNSTILE, we can define "pattern" macros for each type, as in Figure 7. This macro is used exclusively in pattern positions and it matches on, but hides, a type's internal representation.[7] While this feature is not strictly necessary, it relieves some notational burden for programmers implementing "dependent type systems as macros."

```
(define-m (~→vid [x : τin] ... τout) (#%app →̅vid (λ (x ...) τin ... τout)))
```

Fig. 7. Pattern matching macro for the $\rightarrow_{\text{vid}}$ type.

3.4 Putting It All Together

We've now seen all the components necessary to define dependent types as macros: a `struct` record declaration for the internal representation, a `define-tyrule` implementing the rule for type construction and elaborating to the `struct`, and one or more pattern macros. As a convenience, we add a new construct to TURNSTILE, `define-type`, that automatically generates the boilerplate and allows language implementors to simply write down the rule for well-formed dependent types.

In Figure 8, we show how to implement (a simplification of) the typing rules for TYPED VIDEO using the approach we've presented, and our extensions to TURNSTILE. We see that integer terms may be lifted to the type level via the `Producer` type constructor. The new lambda rule resembles the rule for $\rightarrow_{\text{vid}}$ from Figure 4, except a lambda has the $\rightarrow_{\text{vid}}$ type. Similarly, the application rule requires that the type of its operator matches a $\rightarrow_{\text{vid}}$ type, and that its argument has the type of the $\rightarrow_{\text{vid}}$ type's inputs.

The lambda rule is a multi-clause `define-tyrule`, analogous to multi-clause macros, because we only wish to allow lifting of integer terms to the type level. Notice that the first clause identifies `Int` cases with the `~Int` pattern macro generated by `define-type`. When the parameter does not have integer type, type checking falls through to the second clause, where the output $\rightarrow_{\text{vid}}$ type is constructed with a fresh `dummy` name, so it may not be referenced in subsequent types. As expected, in the output of the $\texttt{app}_{\texttt{vid}}$ type rule, we substitute references to the binder in $\overline{\tau}_{\text{out}}$ with the argument from the application.

3.5 Type-Level Computation

Since types may contain integer expressions, we must add type-level computation to normalize the types thus the integer constraints. We present two approaches to type-level computation "as macros": a simple approach here, and a more modular and extensible approach in Section 4.

To enable the first approach, we again modify the "dependent types as macros" core API. First, we add a new interposition point in the `assign` metafunction, so the new definition is the following.

```
  (define (assign e τ) (attach e 'type (τ-eval τ)))
```

The interposition point $\boxed{\tau\texttt{-eval}}$ enables customization of type normalization. Since `assign` is implicitly called in the conclusion of every `define-tyrule`, interposing on `τ-eval` allows us to inject

---

[7] The pattern macros could have the same name as its analogous type, but to better distinguish pattern positions we (and TURNSTILE) follow Racket's convention of prefixing pattern macros with ~. See Figure 8 for a usage of a pattern macro.

typed/video

```
#lang turnstile
(provide (rename [λ_vid λ] [app_vid #%app])
(define-type Int : Type)
(define-type Producer : Int -> Type)
(define-type →_vid #:binders ([X : Type]) : Type)
(define-tyrule λ_vid
 [(_ [x : τ_in] e) ≫ ; Int case
  [⊢ τ_in ≫ ~Int ⇐ Type]
  [[x ≫ x̅ : Int] ⊢ e ≫ e̅ ⇒ τ_out]
  ----------------------------
  [⊢ (λ (x̅) e̅) ⇒ (→_vid [x̅ : Int] τ_out)]]
 [(_ [x : τ_in] e) ≫
  [⊢ τ_in ≫ τ̅_in ⇐ Type]
  [[x ≫ x̅ : τ̅_in] ⊢ e ≫ e̅ ⇒ τ_out]
  ----------------------------
  [⊢ (λ (x̅) e̅) ⇒ (→_vid [dummy : τ̅_in] τ_out)]])
(define-tyrule (app_vid f e) ≫
  [⊢ f ≫ f̅ ⇒ (~→_vid [x̅ : τ̅_in] τ̅_out)]
  [⊢ e ≫ e̅ ⇐ τ̅_in]
  ------------------------------------
  [⊢ (#%app f̅ e̅) ⇒ (subst e̅ x̅ τ̅_out)])
```

Fig. 8. Typed Video type definitions, lambda, and function application rules.

typed/video

```
#lang turnstile
(define-[τ-eval]
 [n^int n] [b^bool b]
 [(+ n m) #:with n*^int ([τ-eval] n) #:with m*^int ([τ-eval] m) (+ n*.val m*.val)]
 [(< n m) #:with n*^int ([τ-eval] n) #:with m*^int ([τ-eval] m) (< n*.val m*.val)]
 [(Producer n) (Producer ([τ-eval] n))]
 [other other])
```

Fig. 9. Excerpt of type-level evaluation in the Typed Video language.

the needed behavior. By default, `τ-eval` just expands a type; for Typed Video, we implement an interpreter for the index language. Figure 9 shows a (simplified version of) this function. We use `define-`$\boxed{\texttt{τ-eval}}$ to redefine the $\boxed{\texttt{τ-eval}}$ function used by other type rules. The definition is a series of pattern-body clauses. When `τ-eval` is called with a type τ, the first clause whose pattern matches τ is used. The first two clauses match literal values. The third clause matches on addition. This clause first recursively calls `τ-eval` on the arguments. If evaluating those terms produce syntactic literal numbers, then the actual arithmetic operation is performed. This fourth case is similar. If the input to `τ-eval` is a `Producer`, then its index is evaluated, otherwise the type is left unchanged.

4 A DEPENDENTLY-TYPED CALCULUS

The approach to type-level computation for the Typed Video language in Section 3 suffices when the index language is simple. It does not scale well when, for example, we want to define new reduction rules that can be used both for run-time and during type checking, as is common in type theory. This section presents a more general, extensible approach to adding type-level computation via macros where types and terms may mix.

Again, we do this in the context of an example language. We start with essentially the Calculus of Constructions (CC) [Coquand and Huet 1988], which features "heavyweight", also called full-spectrum, dependent types, in which there is no distinction between terms and types. We gradually extend our initial implementation with type schemas, ala Martin-Löf Type Theory [Martin-Löf 1975], and finally extend to the Calculus of Inductive Constructions [Pfenning and Paulin-Mohring 1989]. This demonstrates that our approach scales to the same calculi used in contemporary proof assistants. Each extension is entirely modular: it does not requiring modifying any prior code, and only defines new macros. This demonstrates key features of the "dependent type systems as macros" approach: modularity and extensibility.

We start by upgrading Figure 2's simply-typed language into CC by:

1. changing the → type into a Π type, whose output type can refer to its input type;
2. modifying the lambda and application rules to introduce and eliminate the Π type; and
3. implementing reduction rules for type-level computation.

We first present the key concepts as they apply to macro systems in general, and then the new Turnstile abstractions that support on-paper notation.

4.1 Defining Type-Level Reductions

Figure 10 presents dep-lang, a dependent calculus with Π types, *i.e.*, dependently typed functions. The new lambda rule introduces the Π type and the function application rule eliminates it. The key difference from Typed Video's calculus is in the conclusion of the function application rule.

```
dep-lang

#lang turnstile
(provide Π (rename [λ_dep λ] [app_dep #%app])
(define-type Π #:binders ([X : Type]) -> Type)
(define-tyrule (app_dep f e) ≫
  [⊢ f ≫ f̅ ⇒ (~Π [X̅ : τ̅_in] τ̅_out)]
  [⊢ e ≫ e̅ ⇐ τ̅_in]
  ------------------------------------
  [⊢ (β f̅ e̅) ⇒ (⇑/v1 (subst e̅ X̅ τ̅_out))])

(define-tyrule (λ_dep [x : τ] e) ≫
  [⊢ τ ≫ τ̅ ⇐ Type]
  [[x ≫ x̅ : τ̅] ⊢ e ≫ e̅ ⇒ τ̅_out]
  ------------------------------
  [⊢ (λ (x̅) e̅) ⇒ (Π [x̅ : τ̅] τ̅_out)])
```

Fig. 10. A dependently-typed lambda calculus.

In app$_{\text{dep}}$'s output type, we replace the Π type binder x̅ with the argument of the application e̅. To support arbitrary run-time terms in the type system, the type is also wrapped with a "reflect" operation `⇑/v1` that will be explained in detail shortly. To implement the reduction rule for Π, the output "term" (which is also a type) is wrapped with a β macro which implements the reduction rule for Π, enabling evaluation during type checking.

Figure 11 defines the β-reduction rule as a macro. The β macro expands its head expression and matches on that result using an explicit `syntax-parse` syntax pattern matcher. If the expanded head matches a λ (first case), occurrences of the λ parameter `x` in the body are replaced with the argument e. Performing the reduction, however, may create additional redexes, for example if the argument itself is a function. To further reduce these new redexes in the contractum, we need to "reflect" references to run-time representations `#%app` back to β, and the `⇑/v1` function performs this operation. Otherwise (second case), the result is an unreduced run-time `#%app` term.

While Figure 11 conceptually captures our approach to type-level computation via macros, this obvious implementation is not extensible since the reflection operation would need to know about all possible reduction rules in advance. Instead, we add a new core API function ⇑ for reflection, defined in Figure 12, which is extensible via annotation on syntax objects. Instead of just replacing

```
(define-m (β f e)
  (syntax-parse (local-expand f)
   [(λ (x) body) (⇑/v1 (subst e x body))]
   [f̅ (#%app f̅ e)]))

(define (⇑/v1 e) (subst β #%app e))  ; fn mapping #%app back to β; not extensible
```

Fig. 11. Beta reduction rule, implemented as a plain macro.

`#%app`, it traverses a piece of syntax and checks for the syntax property *'reflected-name*. If the property exists, the value associated with the key is used as the reflected name. The `mk-reflected` function expects a *placeholder*, an identifier to use as the run-time representation of the elimination form, and attaches a given identifier for use as the *'reflected-name* property. For example, the *placeholder* for application is `#%app`, and the reflected id will be the β macro.

```
(define (⇑ e)  ; extensible fn mapping terms to surface stx
  (syntax-parse e
    [placeholder^id (detach placeholder 'reflected-name)]
    [(e ...) ((⇑ e) ...)]
    [else e]))

(define (mk-reflected placeholder reflected-id)
  (attach placeholder 'reflected-name reflected-id))

; example: ((mk-reflected #%app β) (λ (x) x) (λ (x) x)) = (#%app (λ (x) x) (λ (x) x))
; example: (⇑ (mk-reflected #%app β)) = β
; example: (⇑ ((mk-reflected #%app β) (λ (x) x) (λ (x) x))) = (β (λ (x) x) (λ (x) x))
; example: (local-expand (⇑ ((mk-reflected #%app β) (λ (x) x) (λ (x) x)))) = (λ (x) x)
```

Fig. 12. Core API for reflection

This API, while small involves a fundamental change in how macro expansion proceeds. Typically, we think of the work flow with macros as: 1) macro expansion, 2) runtime evaluation. In the previous "type systems as macros" work, this changes to: 1) macro expansion + type checking (interleaved) 2) runtime. Macro expansion is interleaved with type checking, since the type system is defined in macros. This requires communicating types between macros, which happens through syntax properties on syntax objects. With our extension to the "types systems as macros" API, this changes again. Instead, we have: 1) macro expansion + type checking + evaluation (interleaved), 2) runtime evaluation. All are mutually recursive and we must coordinate information between each stage. Now we must communicate how a reduced term corresponds to a type (*i.e.*, macro), and therefore, to its own type-level reduction semantics; this is the role of the reflection API above. We describe this interleaved semantics and the requirements it imposes on macros systems in Appendix B.

For our Turnstile implementation, we add abstractions to avoid the boilerplate in the pattern described above. Figure 13 defines `define-red`, a macro-defining macro. Given a redex and contractum, it generates macros like β in Figure 11, where the generated macro automatically handles reflecting the contractum with ⇑. In essence, multiple `define-red` declarations cooperate with each other through the API in Figure 12.

The definition of `define-red` is a multi-clause macro, which itself generates a macro representing a reduction rule, such as β, but automatically inserts the ⇑ and `mk-reflected` calls as required. The

```
(define-m define-red  ; Turnstile form for defining reduction rules
  [(define-red red-name redex ~> contractum)  ; single-redex case
   (define-red red-name [redex ~> contractum])]  ; rewrite to match second case
  ; multi-redex case
  [(define-red red-name [(placeholder redex-hd redex-rst ...) ~> contractum] ...)
   ⌜(define-m (red-name hd arg ...)                                ⌝
      (syntax-parse ((local-expand hd) arg ...)
       [(redex-hd redex-rst ...) (⇑ contractum)] ...
   ⌞   [(e ...) ((mk-reflected placeholder red-name) e ...)]))⌟])
```

Fig. 13. Turnstile form for defining reduction rules.

```
(define-red β (#%app (λ (x) body) e) ~> (subst e x body))
```

Fig. 14. Beta reduction rule, implemented with `define-red`.

first case is short-hand for easy reduction rules, and recursively calls `define-red` to invoke the second case. The second case accommodates multiple redexes and contractums. (The output of the second clause is marked with ⌜⌟ instead of the usual blue text color, to avoid obscuring nested patterns and templates in the generated macro.) The generated reduction macro, `red-name`, behaves essentially like a generalized version of β in Figure 11. More specifically, `red-name` expands the head, and if it matches the supplied redex, rewrites it to the specified contractum. Otherwise, it expands to a term represented by the placeholder but marked with reflection property *’reflected-name*. If further evaluation (*i.e.*, macro-based reductions) causes the term to become redex, then ⇑ ensures that `red-name` is invoked again to reduce the redex. Figure 14 shows the definition of the β-reduction rule implemented with Turnstile’s `define-red`, where the redex is a macro pattern from the underlying macro system and the contractum rewrites components of that pattern. This definition concisely matches how such a rule would be written in a textbook.

4.2 A Little Sugar

The dep-lang language from Figure 10 and Figure 14 is equivalent to the Calculus of Constructions (CC) [Coquand and Huet 1988]. There are many tutorials on implementing dependent types, and they typically end here, but it is very difficult to actually program or prove with CC. Fortunately, a fundamental feature of the “(dependent) type system as macros” approach is that DSLs gain extensibility via macros, for free. We demonstrate this for dependent types in Figure 15, which presents dep-lang/sugar, a library that adds syntactic sugar for dep-lang using macros—local, modular elaboration passes. In contrast, a typical implementation of a dependently-typed language would add a whole-program elaboration pass on top of the core. We subsequently use our dep-lang/sugar library to add additional type schemas.

We define automatically-currying, multiple-argument versions of Π, λ, and function application. We may also define $\rightarrow$ and $\forall$ as shorthands for Π, where the former generates an arbitrary name, and the latter inserts implicit `Type` annotations. These new variations are exported with the same name as their single-arity versions, using the interposition feature of the macro system to interpose on the definitions from dep-lang. Thus new dep-lang programs importing this library will automatically use the new sugary forms.

The last macro in our library, `define-data-constructor`, wraps `define-type` with the just-defined λ/c to allow partial application of its constructor. Like `define-type`, `define-data-constructor` supports syntax for declaring data structures with either named or unnamed arguments.

```
#lang dep-lang                                                dep-lang/sugar
(provide → ∀ (rename [λ/c λ] [app/c #%app] [Π/c Π]))
(define-m Π/c
  [(_ e) e]
  [(_ x . rst) (Π x (Π/c . rst))])
(define-m (→ τ_in ... τ_out) (Π [TMP : τ_in] ... τ_out)) ; TMP fresh
(define-m (∀ X ... τ) (Π [X : Type] ... τ))
(define-m λ/c
  [(_ e) e]
  [(_ x . rst) (λ x (λ/c . rst))])
(define-m app/c
  [(_ e) e]
  [(_ f e . rst) (app/c (#%app f e) . rst)])

(define-m define-data-constructor
 [(define-data-constructor name : τ ... → τ_out)
  (define-data-constructor name : [TMP : τ] ... → τ_out)] ; TMP fresh
 [(define-data-constructor name : [x : τ] ... → τ_out)
 ⌜(define-type name : [x : τ] ... -> τ_out)        ⌝
  (define-m name (λ/c [x : τ] ... (name x ...)))
 ⌞(define-m ~name ~name)                           ⌟)
```

Fig. 15. A DEP-LANG library that adds some syntactic sugar, e.g., currying.

4.3 A Library of Natural Numbers

Technically, we could Church-encode all our programs and proofs, but this is somewhat impractical. Luckily, we have already developed all tools to extend DEP-LANG with new datatypes. Figure 16 extends DEP-LANG with a natural number library, using a type schema in the style of Martin-Löf Type Theory [Martin-Löf 1975]. Each type schema defines a type, an introduction rule, and an elimination rule; we ignore equivalence rules for this presentation. Specifically, we define `Nat` using the TURNSTILE `define-type` form. We use the `define-data-constructor` variant of `define-type` (from Figure 15) to define the standard introduction rules, `Z` and `S`, corresponding to "zero" and "successor". The elimination form, `elim`$^{\text{Nat}}$, corresponds to a fold over the datatype. Following the terminology of McBride [2000], the form (`elim`$^{\text{Nat}}$ `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 the successor of a number. 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`$^{\text{Nat}}$, one each for `Z` and `S`, as succinctly as in a textbook. Observe that the pattern macros `~Z` and `~S`, defined by `define-type`, are useful when specifying the reduction. For convenience, the DEP-LANG/NAT library also extends `#%datum`, an interposition point for interpreting literal data. With `new-datum`, users of the DEP-LANG/NAT library can write numeric literals in place of the more cumbersome `Z` and `S` constructors. The last `new-datum` clause falls back to the current `#%datum`, making this library compatible with other literal data. We can even support diamond extensions by importing *two* existing versions of `#%datum` (under different names) and using them in separate `new-datum` clauses and, of course, writing some macros to automate such boilerplate.

dep-lang/nat

```
#lang dep-lang
(provide Nat Z S elim^Nat (rename [new-datum #%datum]) +)

(define-type Nat : Type)
(define-data-constructor Z : Nat)
(define-data-constructor S : Nat → Nat)

(define-tyrule (elim^Nat n P mz ms) ≫
  [⊢ n ≫ n̄ ⇐ Nat] ; target
  [⊢ P ≫ P̄ ⇐ (→ Nat Type)] ; prop / motive
  [⊢ mz ≫ m̄z̄ ⇐ (P̄ Z)] ; method for Z
  [⊢ ms ≫ m̄s̄ ⇐ (Π [k : Nat] (→ (P̄ k) (P̄ (S k))))] ; method for S
  -----------
  [⊢ (eval^Nat n̄ P̄ m̄z̄ m̄s̄) ⇒ (P̄ b̄)])

(define-red eval^Nat
  [(elim^Nat ~Z P mz ms) ~> mz]
  [(elim^Nat (~S k) P mz ms) ~> (ms k (eval^Nat k P mz ms))])

(define-m new-datum
  [(new-datum n^nat) #:when (zero? n) Z]
  [(new-datum n^nat) (S (new-datum (- n 1)))]
  [(new-datum x) (#%datum x)])

(define + ; implements n + m
  (λ [n : Nat]
    (elim^Nat n
      (λ [m : Nat] (→ Nat Nat))
      (λ [m : Nat] m)
      (λ [n-1 : Nat] [ih : (→ Nat Nat)] (λ [m : Nat] (S (ih m)))))))
```

Fig. 16. A DEP-LANG library for natural numbers.

4.4 An Equality Type Library, and Applying Telescopes

Figure 17 shows an implementation of the standard equality, or identity, type. The `elim`$^{=}$ rule resembles `elim`$^{\text{Nat}}$ from Figure 16: for any motive `P` such that `(P a)` holds, eliminating a proof that `a = b` allows concluding that `(P b)` holds.

The implementation of `elim`$^{=}$ demonstrates the implicit support for *telescopes* we've added to TURNSTILE to simplify implementing dependent types. The arguments are named and subsequent arguments may reference previous names. Note that *checking* a telescope and *applying* a constructor with telescoping arguments, which involves substitution, are two distinct operations. Section 3.2 presented our implementation of abstractions for the former; the rest of this subsection addresses the latter with a novel, pattern-based substitution technique for instantiating types in a telescope.

Figure 18 shows the relevant parts of `define-type`, which generates a `define-tyrule` that uses this technique. `define-type` first validates the $\kappa_{\text{in}} \ldots \kappa_{\text{out}}$ annotations supplied by the programmer, using the new `folding-check` from Section 3.2. The conclusion (a ≫ variant accommodates emitting definitions) produces the type rule for constructing `name` types.

The key is the reuse of the $\overline{\mathtt{A}}$ ... 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, $\overline{\mathtt{A}}$ is bound to the arguments supplied to that constructor. Since $\overline{\mathtt{A}}$ binds references in $\overline{\kappa_{\text{in}}}$ ..., use of $\overline{\kappa_{\text{in}}}$ ... in

```
#lang dep-lang                                                    dep-lang/eq
(provide = refl elim⁼)

(define-type = : [A : Type] [a : A] [b : A] → Type)
(define-data-constructor refl : [A : Type] [e : A] → (= A e e))

(define-tyrule (elim⁼ t P pt w peq) ≫
  [⊢ t ≫ t̅ ⇒ A]
  [⊢ P ≫ P̅ ⇐ (→ A Type)]
  [⊢ pt ≫ p̅t̅ ⇐ (P̅ t̅)]
  [⊢ w ≫ w̅ ⇐ A]
  [⊢ peq ≫ p̅e̅q̅ ⇐ (= A t̅ w̅)]
  --------------
  [⊢ p̅t̅ ⇒ (P̅ w̅)])
```

Fig. 17. A DEP-LANG library for the equality type.

the output syntax template *automatically* instantiates any $\overline{\mathtt{A}}$s in $\overline{\kappa_{\mathtt{in}}}$ with the concrete arguments to the `name` type constructor, which is the desired behavior. In other words, we hijack substitutions that the macro system already performs with pattern variables in templates to instantiate type variables. The technique is safe, *i.e.*, no variables are captured, thanks to hygiene.

```
(define-tyrule (define-type name : [Aⁱᵈ : κin] ... → κout) ≫
  [[A ≫ A̅ : κin ≫ κ̅i̅n̅ ⇐ Type] ... ⊢ κout ≫ κ̅o̅u̅t̅ ⇐ Type]
  -------------------------------------
  [≻ ⌜(define-tyrule (name A̅ ...) ≫⌝
        [⊢ A̅ ≫ A̿ ⇐ κ̅i̅n̅] ...
        --------------------
        [⊢ (n̅a̅m̅e̅ A̿ ...) ⇒ κ̅o̅u̅t̅])
     ⌞ ; rest of the macro elided          ⌟])
```

Fig. 18. (Part of) the implementation of `define-type`.

With numbers and equality, we can now write a simple example proof in DEP-LANG. Figure 19 proves the additive identity of the natural numbers. The left identity is simple since + is defined by recursion on its first argument and `(+ 0 n)` trivially reduces to `n`. The right identity requires an argument by induction, since `(+ n 0)` cannot evaluate until we know more about `n`.

4.5 Indexed Inductive Type Families

TURNSTILE easily supports type schemas, but type schemas are a modification to the trusted core. Realistic proof assistants instead support safe extension through inductively-defined type families Dybjer [1994]. Inductive types can be implemented in one set of general-purpose rules that are proven sound, and then users can declare new inductive types without extending the trusted core. Adding indexed inductive type families is straightforward using the constructs we have already presented.

Figure 21 presents `define-datatype`, which enables defining inductive types. Our version is based on Brady's presentation of TT [Brady 2005]. The complete implementation requires 18 lines of

```
#lang dep-lang                                                        dep-lang-prog
(require dep-lang/nat dep-lang/eq)
(ann (λ [n : Nat] (refl Nat n)) : (Π [n : Nat] (= Nat (+ 0 n) n)))
(ann (λ [n : Nat]
      (elim^Nat n
        (λ [m : Nat] (= (+ m 0) m))
        (refl Nat 0)
        (λ [k : Nat] [p : (= (+ k 0) k)]
         (elim^= (+ k 0)
                (λ x (= Nat (S (+ k 0)) (S x)))
                (refl Nat (S (+ k 0)))
                k
                p))))
    : (Π [n : Nat] (= Nat (+ n 0) n)))
```

Fig. 19. An example DEP-LANG program: proving additive identity.

code.[8] It makes DEP-LANG equivalent to the Calculus of Inductive Constructions, *i.e.*, the core of the Coq proof assistant, demonstrating that "dependent type system as macros" scales to expressive type theories, and supports on-paper notation even for advanced typing rules like inductive types.

$$\frac{A : \star \qquad n : \mathbb{N}}{\mathrm{Vec}\ A\ n : \star} \quad \underline{\text{where}} \quad \frac{}{\mathrm{nil} : \mathrm{Vec}\ A\ 0} \quad \frac{k : \mathbb{N} \qquad x : A \qquad xs : \mathrm{Vec}\ A\ k}{\mathrm{cons}\ x\ xs : \mathrm{Vec}\ A\ (\mathrm{S}\ k)}$$

```
#lang dep-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))])
```

Fig. 20. Indexed list data definitions, (Top) by hand, and (Bottom) in DEP-LANG.

To help our explanation of `define-datatype`, we begin with a concrete example. Figure 20 shows two length-indexed list data definitions, the first using a natural deduction style as commonly written in the literature (*e.g.*, [McBride and McKinna 2004]) and the second as written with `define-datatype` in DEP-LANG, which is based on Coq's notation. The main source of complexity compared to previous type definitions is that indexed inductive type families distinguish between *parameters* (the `A` in the figure), and *indices* (the `i` in the figure). Briefly, parameters are invariant across the definition while indices may vary. The key is that in both the formal notation and the code, the rules for the data constructors reference the parameter A that is bound in the type definition, while the index argument is specific to each rule. It turns out that this invariance of parameters can be used to simplify the implementation of `define-datatype`, as the following prose explains.

At a high-level, `define-datatype` is "just" a macro that produces four output definitions:

(1) a `define-type` type definition;
(2) `define-data-constructor` data constructor definitions;
(3) a `define-tyrule` elimination rule; and
(4) a `define-red` reduction rule.

---

[8] Admittedly, we elide positivity checking for simplicity.

```
#lang turnstile                                                                        dep-lang
(define-tyrule (define-datatype TY [A : τ_A] ... : [i : τ_i] ... ⟶ τ
                             [C : [i|x : τ_in] ... ⟶ τ_out] ...) ≫
 [[A ≫ A̅ : τ_A ≫ τ̅_̅A̅ ⇐ Type] ...[[i ≫ i̅ : τ_i ≫ τ̅_̅i̅ ⇐ Type] ... ⊢ τ ≫ τ̅ ⇐ Type]
                           [[i|x ≫ i̅|̅x̅ : τ_in ≫ τ̅_̅i̅n̅ ⇐ Type] ... ⊢ τ_out ≫ τ̅_̅o̅u̅t̅ ⇐ Type]...]
 #:with (TY _ ... τ̅_̅o̅u̅t̅i̅ ...) (τ̅_̅o̅u̅t̅ ...)
 #:with (((i̅_̅r̅e̅c̅ ... x̅_̅r̅e̅c̅) ...) ...) (find-recur (([i̅|̅x̅ τ̅_̅i̅n̅] ...) ...) TY)
 --------------------------
 [≻ ⌜(define-type TY : [A : τ_A] ... [i : τ_i] ... → τ) ; define the type and constructors        ⌝
    (define-data-constructor C : [A : τ_A] ... [i|x : τ_in] ... → τ_out) ...

    (define-tyrule (elim^TY v P m ...) ; eliminator for TY
     [⊢ v ≫ v̅ ⇒ (~TY A̅ ... i_inferred ...)] ; target
     [⊢ P ≫ P̅ ⇐ (Π [i̅ : τ̅_̅i̅] ... (→ (TY A̅ ... i̅ ...) Type))] ; motive
     [⊢ m ≫ m̅ ⇐ (Π [i̅|̅x̅ : τ̅_̅i̅n̅] ...
                 (→ (P̅ i_rec ... x̅_̅r̅e̅c̅) ... (P̅ τ̅_̅o̅u̅t̅i̅ ... (C A̅ ... i̅|̅x̅ ...))))] ...
     ----------------------
     [⊢ (eval^TY v̅ P̅ m̅ ...) ⇒ (P̅ i_inferred ... v̅)])

    (define-red eval^TY ; define reduction rule
  ⌞  [(elim^TY (~C A ... i|x ...) P m ...) ~> (m i|x ... (eval^TY x_rec P m ...) ...)] ...) ⌟])
```

Fig. 21. Implementation of `define-datatype` makes DEP-LANG equivalent to CIC.

But the details are dense so we go line-by-line.

- `(define-tyrule (define-datatype TY [A : τ_A] ... : [i : τ_i] ... ⟶ τ`
  `[C : [i|x : τ_in] ... ⟶ τ_out] ...)`

This defines a new type checking macro named `define-datatype`. The first part of the inputs consists of the name of a new type `TY`, its parameter names `A` ..., the types of those parameters τ_A ..., index names `i` ..., and the types of those indices τ_i .... Together, `[A : τ_A] ... [i : τ_i] ...` is a telescope, where each τ_A ... τ_i ... may reference the names that come before it. The result type of the type constructor `TY` itself is the type τ. The second line of the input specifies the constructors for `TY`, `C` .... The type of the constructors is described by the telescopes `[i|x : τ_in] ...` (where `i|x` is a literal identifier). Finally, a fully-applied constructor has type of shape τ_{out}, which we refine later. The key is that the `A` binders range over the entire declaration, *i.e.*, the τ_{in} and τ_{out} types may also reference `A`, which is not true of the `i` binders.

- `[[A ≫ A̅ : τ_A ≫ τ̅_̅A̅ ⇐ Type] ...[[i ≫ i̅ : τ_i ≫ τ̅_̅i̅ ⇐ Type] ... ⊢ τ ≫ τ̅ ⇐ Type]`
  `[[i|x ≫ i̅|̅x̅ : τ_in ≫ τ̅_̅i̅n̅ ⇐ Type] ... ⊢ τ_out ≫ τ̅_̅o̅u̅t̅ ⇐ Type]...]`

These premises validate that the types supplied by the programmer in a `define-datatype` declaration have type Type. It uses the new folding TURNSTILE syntax introduced in Section 3.2, but with a new twist. Since the `A` ranges over the entire definition, we are essentially checking *nested* telescopes; TURNSTILE supports this, using the slightly altered syntax, compared to Section 3.2, above.

- `#:with (TY _ ... τ_outi ...) (τ_out ...)`

This extracts the index arguments in the data constructor output types and binds them to the τ_{outi}... pattern, which is later used to check the eliminator methods. For example, τ_{outi}... would correspond to `0` and `(S k)` in Figure 20's definitions.

- $\texttt{\#:with (((}\texttt{i}_{\texttt{rec}}\texttt{ ... }\texttt{x}_{\texttt{rec}}\texttt{) ...) ...) (find-recur (([i|x }\tau_{\texttt{in}}\texttt{] ...) ...) TY)}$

  This line finds the recursive arguments for each constructor `C`. The function, `find-recur` (elided), returns arguments $\texttt{x}_{\texttt{rec}} \in \texttt{i|x ...}$ whose type is equal to `TY`. In addition, `find-recur` finds the indices $\texttt{i}_{\texttt{rec}}\texttt{ ...} \subseteq \texttt{i|x ...}$ that the type of $\texttt{x}_{\texttt{rec}}$ references.
- $\texttt{(define-type TY : [A : }\tau_{\texttt{A}}\texttt{] ... [i : }\tau_{\texttt{i}}\texttt{] ... }\rightarrow\tau\texttt{)}$

  This defines type `TY`, using parameters and indices given to `define-datatype`.
- $\texttt{(define-data-constructor C : [A : }\tau_{\texttt{A}}\texttt{] ... [i|x : }\tau_{\texttt{in}}\texttt{] ... }\rightarrow\tau_{\texttt{out}}\texttt{) ...}$

  This line defines data constructors `C ...`, with the type specified in the input to `define-datatype`. It uses the `define-data-constructor` form, from DEP-LANG/SUGAR, so that the constructors may be partially applied. Note that parameters $\texttt{[A : }\tau_{\texttt{A}}\texttt{] ...}$ are added to each constructor declaration.
- $\texttt{(define-tyrule (elim}^{\texttt{TY}}\texttt{ v P m ...)}$

  This implements the type rule for the eliminator named $\texttt{elim}^{\texttt{TY}}$, which has three kinds of inputs: a target `v`, a motive `P`, and methods `m ...`, one for each `C ...`.
- $\texttt{[}\vdash\texttt{ v }\gg\overline{\texttt{v}}\Rightarrow\texttt{(\textasciitilde TY }\overline{\texttt{A}}\texttt{ ... }\texttt{i}_{\texttt{inferred}}\texttt{ ...)]}$

  The target `v` must have type `TY`, with parameters bound to $\overline{\texttt{A}}\texttt{ ...}$ and indices to $\texttt{i}_{\texttt{inferred}}\texttt{ ...}$. This reuse of pattern variables $\overline{\texttt{A}}\texttt{ ...}$ (from the premises to `define-datatype`) is instance of 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 $\overline{\texttt{A}}\texttt{ ...}$, *e.g.*, $\overline{\tau_{\texttt{A}}}$, $\overline{\tau_{\texttt{i}}}$, or $\overline{\tau_{\texttt{in}}}$, will automatically be instantiated with `v`'s parameters by the macro system. Note that we *do not use* this technique for the indices. Instead we bind new pattern variables $\texttt{i}_{\texttt{inferred}}\texttt{ ...}$.
- $\left[\vdash\texttt{ P }\gg\overline{\texttt{P}}\Leftarrow\texttt{(}\Pi\texttt{ [}\overline{\texttt{i}}\texttt{ : }\overline{\tau_{\texttt{i}}}\texttt{] ... (}\rightarrow\texttt{ (TY }\overline{\texttt{A}}\texttt{ ... }\overline{\texttt{i}}\texttt{ ...) Type))}\right]$

  The motive `P` is a function that consumes indices $\overline{\texttt{i}}\texttt{...}$ and a value with type `TY` at those indices, and returns a type that is the result of the elimination. Here $\overline{\tau_{\texttt{i}}}\texttt{...}$ are the types of indexes specified in the input to `define-datatype`, but automatically instantiated with the inferred concrete parameters of the target `v`. The $\overline{\texttt{A}}\texttt{...}$ passed to `TY` are those same parameters.
- $\texttt{[}\vdash\texttt{ m }\gg\overline{\texttt{m}}\Leftarrow\texttt{(}\Pi\texttt{ }\overline{\texttt{[i|x}}\texttt{ : }\overline{\tau_{\texttt{in}}}\texttt{] ...}$
  $\qquad\texttt{(}\rightarrow\texttt{ (}\overline{\texttt{P}}\texttt{ }\overline{\texttt{i}_{\texttt{rec}}}\texttt{ ... }\overline{\texttt{x}_{\texttt{rec}}}\texttt{) ... (}\overline{\texttt{P}}\texttt{ }\overline{\tau_{\texttt{outi}}}\texttt{ ... (C }\overline{\texttt{A}}\texttt{ ... }\overline{\texttt{i|x}}\texttt{ ...))))] ...}$

  A call to the eliminator must include one method for each constructor `C ...`. Each method `m` consumes the constructor inputs $\overline{\texttt{[i|x}}\texttt{ : }\overline{\tau_{\texttt{in}}}\texttt{] ...}$, as specified in the input to `define-datatype`, and an argument for each recursive argument $\overline{\texttt{x}_{\texttt{rec}}}$. These latter arguments represent recursive applications of the eliminators, so have types specified by the motive `P`, *i.e.*, $\texttt{(}\overline{\texttt{P}}\texttt{ }\overline{\texttt{i}_{\texttt{rec}}}\texttt{...}\overline{\texttt{x}_{\texttt{rec}}}\texttt{) ...}$. The type $\texttt{(}\overline{\texttt{P}}\texttt{ }\overline{\tau_{\texttt{outi}}}\texttt{... (C }\overline{\texttt{A}}\texttt{ ... }\overline{\texttt{i|x}}\texttt{ ...))}$ of each method's result is also determined by the motive. The $\overline{\tau_{\texttt{outi}}}\texttt{...}$ comes from the constructor output types specified in the input to `define-datatype`.
- $\texttt{[}\vdash\texttt{ (eval}^{\texttt{TY}}\texttt{ }\overline{\texttt{v}}\texttt{ }\overline{\texttt{P}}\texttt{ }\overline{\texttt{m}}\texttt{ ...) }\Rightarrow\texttt{ (}\overline{\texttt{P}}\texttt{ }\texttt{i}_{\texttt{inferred}}\texttt{ ... }\overline{\texttt{v}}\texttt{)])}$

  The eliminator output calls reduction rule $\texttt{eval}^{\texttt{TY}}$ to reduce redexes where $\overline{\texttt{v}}$ is a fully-applied constructor. Its type is determined by the motive applied to the indices of the target $\overline{\texttt{v}}$ and $\overline{\texttt{v}}$ itself.
- $\texttt{(define-red eval}^{\texttt{TY}}$
  $\quad\texttt{[(elim}^{\texttt{TY}}\texttt{ (\textasciitilde C A ... i|x ...) P m ...) \textasciitilde> (m i|x ... (eval}^{\texttt{TY}}\texttt{ }\texttt{x}_{\texttt{rec}}\texttt{ P m ...) ...)] ...))}$

  The last definition produced by a `define-datatype` declaration is a reduction rule consisting of a series of redexes, one for each constructor `C ...`. The rule states that elimination of a fully-applied constructor `C` reduces to application of the method for that constructor, where the recursive arguments to the method are additional invocations of the eliminator on the recursive constructor arguments. Observe how the macro system's pattern language naturally associates each `C` with its method `m`, again leading to concise definition that matches what language designers write on paper. For comparison, see the specification of inductive type families from Brady [2005].

5 CUR: A PROOF ASSISTANT AS MACROS

To demonstrate that our approach scales to a realistic language, we implemented a prototype proof assistant called Cur. Proof assistants based on dependently-typed languages are unusual in that they directly implement a formal calculus—typically intended as only a theoretical model of computation—as their core language, thus ensuring a small, consistent, trusted computing base. Cur's core is the dep-lang dependent calculus presented in Section 4. To make programming and proving in a calculus practical, proof assistants typically build separate layers of features and DSLs, such as unification for generating annotations, notational support to generating definitions, or tactic systems for constructing proofs. By building Cur with macros from the beginning, we already have a framework in which we, and Cur users, can easily build such DSLs. Further, *all* new DSLs are integrated into the language, instead of as third-party preprocessors.

This section presents two such DSLs: Olly—a DSL for modeling programming languages inspired by Ott [Sewell et al. 2007]—and ntac— a tactic language for scripting proofs. These DSLs elaborate to core Cur during macro expansion, but before type checking, and thus we are able to extend the functionality of our language yet keep the trusted base small. We demonstrate how these DSLs simplify formal development by allowing users to express programs and proofs using familiar notation, rather than the syntax of dependent type theory.

5.1 Olly

Olly is an Ott-like [Sewell et al. 2007] DSL for modeling programming languages in Cur. Specifically, programmers may write BNF notation or inference rule notation to specify language syntax and relations, respectively, and Olly automatically generates the inductive type definitions to represent them. Both notations support extracting the models to LaTeX and Coq, in addition to using the models directly in Cur. Unlike Ott, however, which is an external tool chain, Olly is a user-written library for Cur. As such, it can take advantage of the existing elaboration framework, and is integrated into the standard Cur development environment and language.

Figure 22 shows how one may define the syntax of a simply-typed λ-calculus using Olly. This language includes booleans, unit, pairs, and functions. The definition uses standard BNF notation, with optional annotations of the form `#:bind <var>` to specify a binding position. Note that the `let` form eliminates pairs in this language, and thus binds two names.

olly-prog

```
#lang cur
(require cur/olly)
(define-language stlc  #:vars (x)
 #:output-coq "stlc.v" #:output-latex "stlc.tex"
 val  (v)   ::= true false unit
 type (A B) ::= boolty unitty (-> A B) (* A A)
 term (e)   ::= x v (lambda (#:bind x : A) e) (app e e)
                    (cons e e) (let (#:bind x #:bind x) = e in e))
```

Fig. 22. An STLC example using Olly, a notation extension for Cur.

The first argument, `stlc`, is the language name. The next three are optional arguments: `#:vars` specifies meta-variables for variables in the syntax; `#:output-coq` specifies a Coq output file; and `#:output-latex` specifies a file for a LaTeX rendering of the BNF grammar. After the optional arguments, an arbitrary number of non-terminal definitions are specified.

The `define-language` form generates an inductive type definition for each non-terminal. It uses the language name and the non-terminal names to generate the inductive type and the constructors. For example, below is the definition generated for the `term` non-terminal.

```
(define-datatype stlc-term : Type
  (Var->stlc-term : Var → stlc-term)
  (stlc-val->stlc-term : stlc-value → stlc-term)
  (stlc-lambda : stlc-type stlc-term → stlc-term)
  (stlc-app : stlc-term stlc-term → stlc-term)
  (stlc-cons : stlc-term stlc-term → stlc-term)
  (stlc-let : stlc-term stlc-term → stlc-term))
```

It has a constructor for each kind of `term`. In addition, a conversion constructor is produced for references to other non-terminals, *e.g.*, `Var->stlc-term`. Internally, `define-language` uses an intermediate Racket data structure to represent the grammar, which may then be converted to CUR, Coq, LaTeX, and other outputs. Since extensions are supported linguistically, programmers may use OLLY forms alongside normal CUR code, rather than switch to an external tool.

OLLY demonstrates how "dependent types as macros" supports domain-specific modeling—here, the domain is programming language theory. By starting from macros, the proof assistant is extensible with domain-specific support *by default*, rather than as an after thought. We can tailor all aspects of the proof assistant, from the object theory to the syntax, to our domain.

5.2 A Tactic Language

Tactic systems are a popular addition to proof assistants to enable interactive, command-based construction of proof terms; some proof assistants even feature multiple tactic languages. A tactic system also exercises many interesting features of the elaboration system—they require pre-type-checking-time general purpose computation, traversal and pattern matching of object language terms, interesting data structures in the elaboration system for manipulating proof states, an API to the object language in order to type check and evaluate the terms while constructing proofs, interactivity, and syntactic integration into the language. A macro system, such as Racket's, provides all but the API to the object language, but by developing the type system as macros, we get this meta-language API to the object language by construction.

For a simple example, in Figure 23, we present a hypothetical library of propositional logic, along with a tactic, `tauto`, that automatically finds a term that proves the given type. The tactic is simply a macro that traverses and pattern matches on terms. It uses the pattern combinators generated by `define-datatype`, *e.g.*, `~True`, along with the backtracking inherent in the matching algorithm, to concisely specify the proof search. For more complex tactics, however, we require slightly more than plain macros, *e.g.*, to help maintain proof state and track intermediate theorems.

Thus, we create NTAC, a tactic language for CUR. The NTAC tactic system builds on the basic idea of tactics as macros, but uses a zipper data structure to allow navigation of the proof term and to track the program context. Rather than plain macros, NTAC tactics are host-language (Racket) functions over this zipper that are executed by the macro system during elaboration. While the details of NTAC's navigation and construction of proof terms are not particularly novel as far as tactic systems go, the use of macros allowed us to easily develop the tactic system and integrate it into CUR.

To demonstrate how NTAC integrates into CUR, here is an NTAC proof for a trivial theorem.

tactic-eg

```
#lang cur
(require cur/ntac)
(ntac (forall (A : Type) (a : A) A)
  (by-intros A a)
```

```
(define-type False : Type)

(define-datatype True : Type
 [I : True])

(define-datatype And [X : Type] [Y : Type] → Type
 [conj : [x : X] [y : Y] → (And X Y)])

(define-datatype Or [X : Type] [Y : Type] → Type
 [or-introL : [x : X] → (Or X Y)]
 [or-introR : [y : Y] → (Or X Y)])
```

```
(define-m tauto
  [(tauto ~True) I]
  [(tauto (~And X Y))
   #:with x (tauto #'X)
   #:with y (tauto #'Y)
   (conj X Y x y)]
  [(tauto (~Or X Y))
   #:with x (tauto #'X)
   (or-introL X Y x)]
  [(tauto (~Or X Y))
   #:with y (tauto #'Y)
   (or-introR X Y y)]
  [(~fail "no proof") _])
```

Fig. 23. A library for type-level propositions.

```
  (by-assumption))
```

The `ntac` form builds an expression given an initial goal, *e.g.*, the polymorphic identity type, and a tactic script. It is similar to Coq's `Goal`, which introduces an anonymous goal that can be solved using an Ltac script. Unlike `Goal`, however, `ntac` produces an expression, meaning it can be used in any expression position in Cur, not just as a top-level command. This is the natural design for `ntac`, since macros are naturally extensions to the *expression* language. This example uses the `by-intros` tactic, which takes arguments representing names to bind as assumptions in the local proof environment. Then we conclude the proof with `by-assumption`, which takes no arguments and searches the local environment for a term that matches the current goal. We can create a `define-theorem` definition to assign a name to an NTAC script, so it may be used to help with another proof:

tactic-eg

```
#lang cur
(define-theorem id (forall (A : Type) (a : A) A)
  (by-intros A a)
  (by-assumption))
```

A definition `(define-theorem name goal script ...)` is essentially syntactic sugar for `(define name (ntac goal script ...))`.[9] Implementationwise, NTAC builds a proof tree data structure in the metalanguage—*i.e.*, Racket—that can represent partial Cur terms, *e.g.*, terms with holes. The tree is a single *goal* node, representing a completely unknown term of some type. Each tactic is an operation that manipulates this tree, usually by changing the current goal node to a larger subtree that represents a partial term with a new subgoal. Once there are no goals left, the tree is translated into a Cur term. The `ntac` form then, simply applies tactics to this tree:

ntac

```
#lang cur
(define-m (ntac goal tactic ...) (ntac-interp goal (list tactic ...)))
```

The `ntac` form calls `ntac-interp`, which constructs an initial proof tree from `goal` and runs each `tactic`, in order, on the tree. If the resulting tree contains unsolved goals, it raises an error; otherwise, it converts the tree to a Cur term. As an example, we define the `intro` tactic below.

ntac

```
#lang cur
(define-tactic (intro name ctx prooftree)
 (define goal (get-current-goal prooftree))
 (ntac-match goal
```

---

[9] In fact, it is somewhat more complicated to support resugaring and rewriting.

```
  [(~forall (x : A) B)
   (make-apply-node
    goal
    (make-ctxt-node
     (λ (old-ctxt) (dict-set old-ctxt name A))
     (make-new-goal (subst name x B)))
    (λ (body-proof) (λ (name : A) body-proof)))]))
```

This tactic introduces a new variable, `name` when the goal has the shape `(forall (x : A) B)`. The tactic extracts the current goal from the proof tree and pattern matches on it. When the goal matches, we construct a new proof tree using `make-apply-node`. This node describes how to construct a term of type `(forall (x : A) B)`, if it is provided a term `body` of type `B`. It then creates a new subgoal from the type `B` (with references to `x` in `B` replaced with `name`) with `make-new-goal`. This new goal is wrapped with a `make-ctxt-node`, which adds `name` bound to `A` in the environment. When this proof tree is complete, the `ntac-interp` function will apply the Racket function on the last line, `(λ (body-proof) (λ (name : A) body-proof))`, to the completed subtree that has replaced `(make-new-goal B)`. Observe how the coloring denotes the use of quasiquotation to build up the term. More specifically, the outer lambda, produces the inner *syntax object* lambda, except it embeds the variable `name` as its parameter, and `body-proof` as the body.

By using a flexible macro system as the basis for our tactic system, we can even equip user-defined tactics with features like interactivity, as shown in Figure 24 (left). Specifically, the `interactive` tactic uses the `print` tactic to print the proof state, then starts a read-eval-print-loop (REPL). Figure 24 (right) shows an example interactive session. The REPL repeatedly reads in a command and runs it via `run-tactic`; when it sees `quit`, it returns the proof tree.

```
(define-tactic (interactive pt)
  (print pt)
  (match (read-syntax)
    [(quit) pt]
    [tactic
     (interactive (run-tactic pt tactic))]))

(ntac (forall (A : Type) (a : A) A)
  interactive)
```

```
-----------
(forall (A : Type) (forall (a : A) A))
> (by-intro A)
A : Type
-----------
(forall (a : A) A)
....
> by-assumption
Proof complete.
> (quit) ; => < procedure >
```

Fig. 24. (Left) Implementation and use of interactivity tactic; (Right) An interactive proof session.

Our macros-based approach makes it simple to develop a prototype proof assistant capable of realistic proofs. To demonstrate this, we used Cur and NTAC to implement the exercises for several chapters of the *Software Foundations* curriculum [Pierce et al. 2018], totaling several thousand lines of proof scripts.[10] Table 1 presents a list of tactics available in NTAC. The `rewrite` tactics, where most of the work lies, support two versions of the equality type: Coq's default Paulin-Mohring equality, and Martin-Löf's original version. When applied to quantified hypotheses, these tactics will try to automatically instantiate the theorems with a basic search over the current proof state.

`assert`	`intros`	`assumption`	`simpl`	`obvious`	`destruct`
`induction`	`reflexivity`	`interactive`	`rewriteL`	`rewriteR`	`print`

Table 1. List of tactics available in NTAC.

---

[10] https://www.github.com/stchang/macrotypes, https://www.github.com/wilbowma/cur

6 FUTURE WORK

A interesting next step is to experiment with typed tactic DSLs, á la Mtac [Ziliani et al. 2013]'s design. We conjecture that use of Racket's `#lang` framework and TURNSTILE will make it straightforward to do so. We also plan to experiment with automation and integration with other tools, *e.g.*, by calling out to solvers or even using the foreign-function interface during expansion.

Resugaring is another direction for future work. Since type checking is interleaved with macro expansion, some effort is required to prevent abstraction leaks that could expose users to elaborated syntax. For example CUR and NTAC use resugaring during interactive proof sessions. The current resugaring approach is rather ad-hoc, however, but recent advances [Pombrio and Krishnamurthi 2018] could help improve this part of the language, and apply generally to macro-based approaches. Another solution could be to stage expansion to avoid the need for resugaring at all. We are experimenting with "stop lists", *i.e.*, finer-grained knobs for controlling expansion, so that we can maintain the benefits of type checking with macros, but do not expand beyond abstractions that the user cares about during the process.

7 RELATED WORK

Much has been written about **implementing basic dependent types** [Altenkirch et al. 2010; Augustsson 2007; Bauer 2012; Löh et al. 2010; Weirich 2014]. All of these tutorials, however, start from scratch and typically stop short of a practical language. For example, most manually deal with type environments and rely on deBruijn indices for α-equality. Further, they do not include practical features such as user-defined inductive datatypes, and they are not easily extensible with sugar, interactivity, or other companion DSLs that programmers typically need to use with their dependently-typed language. In contrast, we show how our macros-based approach enables both rapid creation of a core dependently-typed language, and scales to a realistic full-spectrum proof assistant with user-defined inductive datatypes and extensible notation.

Extending proof assistants is an active area of research. For example, some dependently-typed languages have explored adding metaprogramming [Brady and Hammond 2006; Christiansen 2014; Devriese and Piessens 2013; Ebner et al. 2017] capabilities. This feature, however, typically requires extending the core language. Other languages like Coq often require writing extensions in a less integrated manner, e.g, programming plugins with OCaml and then linking it with other language binaries. With our approach, we use the metaprogramming facilities inherited from the host language, and thus get to write extensions in a more linguistically supported manner.

One of the most common extensions created by dependently-typed programmers, using many clever methods, is **new tactic languages** [Gonthier and Mahboubi 2010; Gonthier et al. 2011; Krebbers et al. 2017; Malecha and Bengtson 2016]. This suggests that (1) the ability to create domain-specific 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 the approach of Lean [Ebner et al. 2017] or Typed Template Coq [Anand et al. 2018].

8 CONCLUSION

To fully leverage the power of dependent types, programmers should be able to quickly develop their own dependently-typed DSLs with just the right expressiveness for their domain. Further, these DSLs should be easily extensible with any new notation or companion DSLs that might be required to make the language practical for realistic programming. We have demonstrated that a macros-based approach to building dependently-typed DSLs satisfies this criteria. For future work, we hope to leverage the rapid prototyping benefit of our approach to experiment with new type theory features like extensions for parametricity modalities and homotopy type theory, and to leverage the extensibility to further explore other domain-specific dependent type applications.

REFERENCES

2016. RFC: Const-dependent Type System. (June 2016). https://github.com/rust-lang/rfcs/pull/1657

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

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.

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! (2007). http://augustss.blogspot.ru/2007/10/simpler-easier-in-recent-paper-simply.html

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella-Béguelin. 2009. Formal Certification of Code-based Cryptographic Proofs. In *Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/1480881.1480894

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

Benjamin Beurdouche, Franziskus Kiefer, and Tim Taubert. 2017. Verified cryptography for Firefox 57. (July 2017). https://blog.mozilla.org/security/2017/09/13/verified-cryptography-firefox-57/

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.

Adam Chlipala. 2011. Mostly-automated Verification of Low-level Programs in Computational Separation Logic. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation.* 234–245.

Adam Chlipala, Benjamin Delaware, Samuel Duchovni, Jason Gross, Clément Pit-Claudel, Sorawit Suriyakarn, Peng Wang, and Katherine Ye. 2017. The End of History? Using a Proof Assistant to Replace Language Design with Library Design. In *Summit oN Advances in Programming Languages (SNAPL)*. https://doi.org/10.4230/LIPIcs.CVIT.2016.23

David Raymond Christiansen. 2014. Type-Directed Elaboration of Quasiquotations: A High-Level Syntax for Low-Level Reflection. In *of the 26nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL 2014)*. ACM, 1.

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

Ryan Culpepper and Matthias Felleisen. 2010. Fortifying macros. In *Proceeding of the 15th ACM SIGPLAN International Conference on Functional Programming*. 235–246.

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

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.

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

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

Matthew Flatt and PLT. 2010. *Reference: Racket.* Technical Report PLT-TR-2010-1. PLT Design Inc. http://racket-lang.org/tr1/.

Martin Fowler and Rebecca Parsons. 2010. *Domain-Specific Languages.* Addison-Wesley.

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

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

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

Conor McBride and James McKinna. 2004. The View from the Left. *J. Funct. Program.* 14, 1 (2004), 69–111. https://doi.org/10.1017/s0956796803004829

Frank Pfenning and Christine Paulin-Mohring. 1989. Inductively Defined Types in the Calculus of Constructions. In *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings (Lecture Notes in Computer Science)*, Michael G. Main, Austin Melton, Michael W. Mislove, and David A. Schmidt (Eds.), Vol. 442. Springer, 209–228. https://doi.org/10.1007/BFb0040259

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. 2018. Inferring Type Rules for Syntactic Sugar. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 812–825.

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.

Stephanie Weirich. 2014. Pi Forall: notes from OPLSS. (2014). 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

A STYLE AND GLOSSARY

This section summarizes the various style choices used in the paper, and also presents some macro terminology that may help with reading the paper.

A.1 Macros Glossary

- A *syntax object* is the Racket AST representation. It's a tree of symbols, accompanied by context information such as source locations, the program's binding structure, and even arbitrary user-specified metadata.

- *Syntax patterns* deconstruct syntax objects, binding *pattern variables* to different parts of a syntax object, which are themselves syntax objects.
- *Syntax templates* construct syntax objects. They may reference pattern variables, whose corresponding syntax object gets embedded into the constructed syntax object.
- *Syntax properties* are key-value pairs associated with syntax object nodes. We use syntax properties to propagate type and other meta information about a piece of syntax.

A.2 Style

In the paper, to help readability, we often stylize code with colors or abbreviations. This section summarizes a few of our choices.

- `syntax pattern` positions, which deconstruct syntax objects, are highlighted with green. A macro definition's input is frequently a pattern position.
- `syntax template` positions, which construct syntax objects, are in blue. A macro definition's output is frequently a syntax template position. When there are nested positions, e.g., for a macro defining macro, we might instead enclose the syntax template with $\overset{\sqcap}{\sqcup}$, so that the coloring of any nested pattern and syntax template positions are not obscured.
- The name being defined (e.g., a macro, typerule, function, etc.) is always underlined.
- In a pattern, literal symbols to match are marked with bold (latex pmb).
- Pattern variables representing elaborated, i.e., expanded, syntax objects are marked with an overline.
- *Syntax classes*, which additionally constrain the shape of pattern variables, are written with a superscript.

B MACRO SYSTEM FEATURES

This section, beginning with table 2, summarizes the macro system features used in the paper, and their availability in other macro systems. While, to our knowledge, Racket is the only language that combines all the features needed for "type systems as macros", many other popular languages are rapidly adopting the same features in their macro systems.

B.1 Procedural macros

Procedural macros are syntax transformations defined in a general-purpose language supporting arbitrary computations. They are essential to allow arbitrary type-checking logic during expansion.

B.2 Quasiquotation and syntax pattern matching

With quasiquotation, macros construct their expansion using syntax matching the textual form of the language, plus escapes for inserting computed elements. Similarly, macros using syntax pattern matching deconstruct the AST of macro invocations using syntax matching the textual form of the language. We use pattern matching and quasiquotation to give type rule macros relatively readable syntax, even without the Turnstile DSL layer.

B.3 Extensible pattern matching

Turnstile types expand into a common internal representation which enables simple implementations of type equality and substitution. Pattern matching this internal representation is verbose. we use Racket's *pattern expanders*, e.g., in section 3.3, to abstract away the internal representation and create simple pattern forms for each type constructor.

	Racket	Lisp	Clojure	Scala	Rust	Julia	Elixir	Crystal
Procedural macros	X	X	X	X		X	X	
Quasiquotation	X	X	X	X	X	X	X	X
Syntax pattern matching	X			X	X			
Extensible pattern matching	X			X_1				
Automatic hygiene	X				X	X	X	
Syntax properties	X	X_2	X_3	X_4				
Macro-defining macros	X	X	X					X
Identifier macros	X	X	$\sim_5$					
Local expansion	X	X	$\sim_5$					
Interposition points	X							

(1) Referred to as “extractor macros”
(2) Referred to as “property lists”
(3) Referred to as “metadata”
(4) Referred to as “attachments”
(5) Clojure does not have symbol macros or expansion in the environment of local macro definitions, but they can be added as a library. See https://github.com/clojure/tools.macro.

Table 2. Macro system features used in the paper, and their availability in other macro systems.

B.4 Automatic hygiene

Program transformations that introduce temporary variables or references to procedure bindings that may be shadowed need to take care to avoid name capture. Manual solutions include using an operation to generate unique names, and namespace-qualifying references. Macro systems supporting automatic hygiene avoid capture without any such manual intervention. Automatic hygiene is useful for types as macros because it keeps type rules concise. Racket uses the set of scopes hygiene algorithm, which works well with local expansion [Flatt 2016].

B.5 Syntax properties

Syntax properties are key-value pairs that may be attached to syntax during expansion, and communicate extra information about syntax between macros. We annotate typechecked forms with a syntax property in order to communicate the inferred type to the parent form’s expansion.

B.6 Macro-defining macros and identifier macros

In a system supporting macro-defining macros, the expansion of one macro may define another. Identifier macros, also known as symbol macros, allow an identifier to be bound to a macro that expands when the identifier is used in reference position. We use macro-defined identifier macros to cause references to typed variables to be annotated with their type. The Turnstile DSL is also an example of a macro-defining macro, as uses of the `define-tyrule` macro define type rule macros.

B.7 Local expansion

Macro systems with local expansion allow macros to request the expansion of subexpressions, including in the environment of local variable and macro bindings. We use local expansion to typecheck subexpressions and access their inferred type while typechecking the parent expression. Racket’s approach to local expansion is discussed in Flatt et al. [2012].

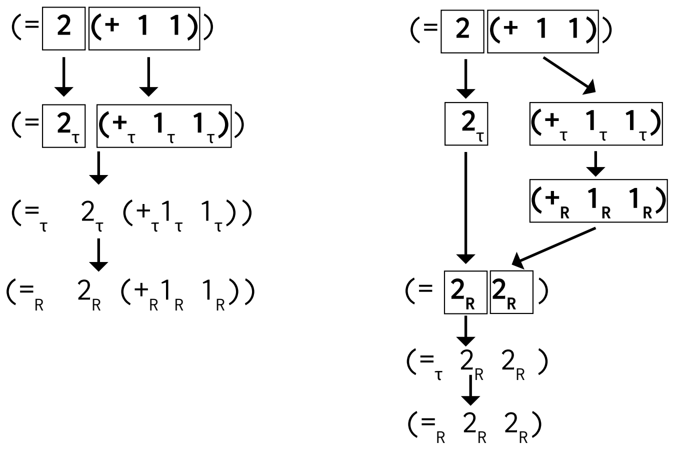

Fig. 25. Example of the interleaving

B.8 Interposition points

Racket's expander automatically inserts hooks at various points, e.g., the `#%app` macro at each function application and the `#%datum` macro at each literal datum (like a number), to allow customizing the behavior of these and other constructs such as modules and the REPL. By redefining these macros, types as macros can add typechecking to the expansion of these syntactic elements.

B.9 Interleaving Semantics

Combined, the above features allow us to interleave macro expansion, type checking, and evaluation, and to communicate information across each stage effectively. In Figure 25 (left), we give an example of a term as it proceeds through the interleaved macro expansion and type checking process, while in Figure 25 (right), we show an example of the interleaving of macro expansion, type checking, and evaluation. Specifically, we show the expansion of an equality type `(= 2 (+ 1 1))`. Without type level reduction, this elaborates each subexpression into the type-annotated version (denoted by the τ subscript), before generating the fully elaborated run-time representations (denoted by the `R` subscript). The type-annotated versions represent the output of type-rule macros, while the run-time representations represent the output of the reduction-rule macros. Without dependent types, the type-annotated and run-time representations are the same. However, once we support dependent types and reduction rules as macros, they are different and require the reflection process described in Section 4.

Notice that, on the right of the figure with reduction during type-checking, we end up with a run-time subterm that must be interleaved with type-annotated terms. Supporting this, particularly when any term (such as a function defined in another module) can be evaluated at expansion time, is the key challenge in the type systems as macros approach, which we solve using many of the above macro system features.