Dependent Type Systems as Macros

Stephen Chang

Northeastern University

November 14, 2019 @ UMass Boston

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 . e : \Pi x : \tau_1 . \tau_2}$$

With collaborators: Alex Knauth, Ben Greenman, Milo Turner, Michael Ballantyne, William Bowman aka,

TURNSTILE+, A Racket-Based Framework for Building Typed Languages

aka,

Let’s Build a Proof Assistant

Overview

1. Introduce macros and macro-based DSLs

2. Introduce type checking via macros

3. Implement a dependently typed core calculus

4. Scale to a full proof assistant ecosystem

Overview

1. Introduce macros and macro-based DSLs
2. Introduce type checking via macros
3. Implement a dependently typed core calculus
4. Scale to a full proof assistant ecosystem

A Macro-based DSL with Racket

UNTYPED-LANG

```
#lang RACKET
(provide-as [new-app #%app]
            [new-λ λ])

(def-stx (new-app f e)
  ; ..
  #'(tgt-app f e))

(def-stx (new-λ x e)
  ; ..
  #'(tgt-λ x e))
```

UNTYPED-PROG

```
#lang UNTYPED-LANG

; uses: new-app, new-λ
```

Module (and language) name

Implementation lang for module

A Macro-based DSL with Racket

UNTYPED-LANG

```
#lang RACKET
(provide-as [new-app #%app]
            [new-λ λ])

(def-stx (new-app f e)
  ; ..
  #'(tgt-app f e))

(def-stx (new-λ x e)
  ; ..
  #'(tgt-λ x e))
```

UNTYPED-PROG

```
#lang UNTYPED-LANG

; uses: new-app, new-λ
```

Module (and language) name

Implementation lang for module

Macro input (pattern match on code)

Macro output (#' generates code)

Defines a macro (code transformer), names are underlined

A Macro-based DSL with Racket

UNTYPED-LANG

```
#lang RACKET
(provide-as [new-app #%app]
            [new-λ λ])

(def-stx (new-app f e)
  ; ..
  #'(tgt-app f e))

(def-stx (new-λ x e)
  ; ..
  #'(tgt-λ x e))
```

UNTYPED-PROG

```
#lang UNTYPED-LANG

; uses: new-app, new-λ
```

- Module (and language) name
- Implementation lang for module
- Exports overload special hooks: e.g., **#%app**
- Macro input (`pattern match on code`)
- Macro output (`#' generates code`)
- Defines a macro (code transformer), names are underlined

A Typed Macro-based DSL

UNTYPED-LANG

```
#lang RACKET
(provide-as [new-app #%app]
            [new-λ λ])

(def-stx (new-app f e)
  ; ..
  #'(tgt-app f e))

(def-stx (new-λ x e)
  ; ..
  #'(tgt-λ x e))
```

UNTYPED-PROG

```
#lang UNTYPED-LANG

; uses: new-app, new-λ
```

TYPED-LANG

```
#lang RACKET
(provide-as [typed-app #%app]
            [typed-λ λ])

(def-stx (typed-app f e)
  ; do type checking
  #'(tgt-app f e))

(def-stx (typed-λ [x : τ] e)
  ; do type checking
  #'(tgt-λ x e))
```

TYPED-PROG

```
#lang TYPED-LANG

; uses: typed-app, typed-λ
```

Overview

1. Introduce macros and macro-based DSLs
2. Introduce type checking via macros
3. Implement a dependently typed core calculus
4. Scale to a full proof assistant ecosystem

A “do type checking” Rule

$$\text{[T-App]}\ \frac{\Gamma \vdash f \Rightarrow \tau_{in} \rightarrow \tau_{out} \qquad \Gamma \vdash e \Leftarrow \tau_{in}}{\Gamma \vdash f\ e \Rightarrow \tau_{out}}$$

Compute type

Check type

Assign type

A “do type checking” Macro

```
(def-stx (typed-app f e)

  ; do type checking

        #'(tgt-app f e)     )
```

Macro input: syntax object

Macro output: syntax object

Macros are “Syntax Object” Transformers

An **S-Expression** is:

- Symbols only
- E.g., `‘(λ x (add1 x))`
  - 1st x and 2nd x unrelated
- `‘(λ x (add1 y))`
  - Is a valid s-expression

A **Syntax Object** (enhanced **S-Expr**) is:

- Symbols
- Binding info
- E.g., `#’(λ x (add1 x))`
  - 1st x binds 2nd x
- `#’(λ x (add1 y))`
  - Is not a valid syntax object (if y free)
- Src Location
- Other arbitrary metadata
  - Types???

“do type checking”

```
(def-stx (typed-app f e)
  ; do type checking
          #'(tgt-app f e)     )
```

“do type checking”

```
(def-stx (typed-app f e)
  ; do type checking
  (attach #'(tgt-app f e) #'τout))
```

Macro output has type information

“do type checking” = expand + attach/detach

```
(def-stx (typed-app f e)
  #:with f⁺ (expand #'f)
  #:with (→ τin τout) (detach #'f⁺)



  (attach #'(tgt-app f⁺ e) #'τout))
```

Macro output has type information

So we can compute a term’s type by expanding

“do type checking” = expand + attach/detach

```
(def-stx (typed-app f e)
  #:with f⁺ (expand #'f)
  #:with (→ τin τout) (detach #'f⁺)
  #:with e⁺ (expand #'e)
  #:with τarg (detach #'e⁺)

  (attach #'(tgt-app f⁺ e⁺) #'τout))
```

Macro output has type information

So we can compute a term’s type by expanding

“do type checking” = expand + attach/detach

```
(def-stx (typed-app f e)
  #:with f+ (expand #'f)
  #:with (→ τ_in τ_out) (detach #'f+)
  #:with e+ (expand #'e)
  #:with τ_arg (detach #'e+)
  #:fail-unless (type= #'τ_arg #'τ_in)
  (attach #'(tgt-app f+ e+) #'τ_out))
```

“do type checking” = expand + attach/detach

```
(def-stx (typed-app f e)
  #:with f⁺ (expand #'f)
  #:with (→ τin τout) (detach #'f⁺)
  #:with e⁺ (expand #'e)
  #:with τarg (detach #'e⁺)
  #:fail-unless (stx= #'τarg #'τin)
  (attach #'(tgt-app f⁺ e⁺) #'τout))
```

“do type checking” = expand + attach/detach

```
(def-stx (typed-app f e)
  #:with f+ (expand #'f)
  #:with (→ τ_in τ_out) (detach #'f+)
  #:with e+ (expand #'e)
  #:with τ_arg (detach #'e+)
  #:fail-unless (stx= #'τ_arg #'τ_in)
  (attach #'(tgt-app f+ e+) #'τ_out))
```

Compute type

Check type

Assign type

$$\text{[T-App]}\ \frac{\Gamma \vdash f \Rightarrow \tau_{in} \to \tau_{out} \qquad \Gamma \vdash e \Leftarrow \tau_{in}}{\Gamma \vdash f\ e \Rightarrow \tau_{out}}$$

Compute type

Check type

Assign type

Turnstile+: Type and Rewrite Rules

$$\frac{\Gamma \vdash f \Rightarrow \tau_{in} \to \tau_{out} \qquad \Gamma \vdash e \Leftarrow \tau_{in}}{\Gamma \vdash f\ e \Rightarrow \tau_{out}}$$

$$\frac{\Gamma \vdash f \Rightarrow \tau_{in} \to \tau_{out} \qquad \Gamma \vdash e \Leftarrow \tau_{in}}{\Gamma \vdash f\ e \Rightarrow \tau_{out}}$$

Conclusion: inputs

```
(def-typed-stx (typed-app f e) »
  [⊢ f » f⁺ ⇒ (→ τin τout)]
  [⊢ e » e⁺ ⇐ τin]
  --------------------------------
  [⊢ (tgt-app f⁺ e⁺) ⇒ τout])
```

Conclusion: outputs

Turnstile+: Type and Rewrite Rules

```
(def-typed-stx (typed-app f e) »
  [⊢ f » f⁺ ⇒ (→ τin τout)]
  [⊢ e » e⁺ ⇐ τin]
  --------------------------------
  [⊢ (tgt-app f⁺ e⁺) ⇒ τout])
```


```
(def-typed-stx (typed-app f e) »
  [⊢ f » f⁺ ⇒ (→ τin τout)]
  [⊢ e » e⁺ ⇐ τin]
  --------------------------------
  [⊢ (tgt-app f⁺ e⁺) ⇒ τout])
```


Desugars to

```
(def-stx (typed-app f e)
  #:with f⁺ (expand #'f)
  #:with (→ τin τout) (detach #'f⁺)
  #:with e⁺ (expand #'e)
  #:with τarg (detach #'e⁺)
  #:fail-unless (stx= #'τarg #'τin)
  (attach #'(tgt-app f⁺ e⁺) #'τout))
```

$\Gamma \vdash e \in T \Rightarrow e'$

[Pierce and Turner 2000, "a four-place type relation … where external language term e yields internal term e′, with type T"]

$$\frac{A \vdash e :: (\tau' \rightarrow \tau) \setminus \bar{e} \qquad A \vdash e' :: \tau' \setminus \bar{e}'}{A \vdash (e\ e') :: \tau \setminus (\bar{e}\ \bar{e}')}$$

[Wadler and Blott 1989, "type and translation rule" For Haskell type classes]

$$\frac{\Gamma \vdash e_1 \rightsquigarrow f_1 : T_1 \qquad \Gamma \vdash e_2 \rightsquigarrow f_2 : T_2 \qquad fun(T_1) = T_{11} \rightarrow T_{12} \qquad T_2 \sim T_{11}}{\Gamma \vdash (e_1\ e_2)^\ell \rightsquigarrow (f_1 : T_1 \Rightarrow^\ell T_{11} \rightarrow T_{12})\ (f_2 : T_2 \Rightarrow^\ell T_{11}) : T_{12}}$$

[Siek, et al. 2015, "gradual typing cast translation"]

TURNSTILE+: Type and Rewrite Rules

```
(def-typed-stx (typed-app f e) »
  [⊢ f » f⁺ ⇒ (→ τin τout)]
  [⊢ e » e⁺ ⇐ τin]
  ------------------------------
  [⊢ (tgt-app f⁺ e⁺) ⇒ τout])
```

```
(def-stx (typed-app f e)
  #:with f⁺ (expand #'f)
  #:with (→ τin τout) (detach #'f⁺)
  #:with e⁺ (expand #'e)
  #:with τarg (detach #'e⁺)
  #:fail-unless (stx= #'τarg #'τin)
  (attach #'(tgt-app f⁺ e⁺) #'τout))
```

TURNSTILE+: Type and Rewrite Rules

```
(def-typed-stx (typed-app f e) »
  [⊢ f » f+ ⇒ (→ τ_in τ_out)]
  [⊢ e » e+ ⇐ τ_in]
  ------------------------------
  [⊢ (tgt-app f+ e+) ⇒ τ_out])
```

```
(def-stx (typed-app f e)
  #:with f+ (expand #'f)
  #:with (→ τ_in τ_out) (detach #'f+)
  #:with e+ (expand #'e)
  #:with τ_arg (detach #'e+)
  #:fail-unless (stx= #'τ_arg #'τ_in)
  (attach #'(tgt-app f+ e+) #'τ_out))
```

TURNSTILE+: Type and Rewrite Rules

```
(def-typed-stx (typed-app f e) »
  [⊢ f » f+ ⇒ (→ τ_in τ_out)]
  [⊢ e » e+ ⇐ τ_in]
  ------------------------------
  [⊢ (tgt-app f+ e+) ⇒ τ_out])
```

```
(def-stx (typed-app f e)
  #:with f+ (expand #'f)
  #:with (→ τ_in τ_out) (detach #'f+)
  #:with e+ (expand #'e)
  #:with τ_arg (detach #'e+)
  #:fail-unless (stx= #'τ_arg #'τ_in)
  (attach #'(tgt-app f+ e+) #'τ_out))
```

TURNSTILE+: Type and Rewrite Rules

```
(def-typed-stx (typed-app f e) »
  [⊢ f » f⁺ ⇒ (→ τ_in τ_out)]
  [⊢ e » e⁺ ⇐ τ_in]
  ----------------------------------
  [⊢ (tgt-app f⁺ e⁺) ⇒ τ_out])
```

```
(def-stx (typed-app f e)
  #:with f⁺ (expand #'f)
  #:with (→ τ_in τ_out) (detach #'f⁺)
  #:with e⁺ (expand #'e)
  #:with τ_arg (detach #'e⁺)
  #:fail-unless (stx= #'τ_arg #'τ_in)
  (attach #'(tgt-app f⁺ e⁺) #'τ_out))
```

TURNSTILE+: Binding Forms

```
(def-typed-stx (typed-λ [x : τin] e) »
  [x » x⁺: τin ⊢ e » e⁺ ⇒ τout]
  ------------------------------
  [⊢ (tgt-λ x⁺ e⁺) ⇒ (→ τin τout)])
```

```
(def-stx (typed-λ [x : τin] e)
  #:with x⁺ (fresh)
  #:with e⁺ (expand #'e #:env #'[x (id-macro (attach x⁺ τin))])
  #:with τout (detach #'e⁺)
  (attach #'(tgt-λ x⁺ e⁺) #'(→ τin τout)))
```

Every variable reference is also a (type rule) macro: expands to a fresh id, with type info

TURNSTILE+: "Type rules" for types

```
(def-typed-stx (→ τin τout) »
  [⊢ τin » τin⁺ ⇐ Type]
  [⊢ τout » τout⁺ ⇐ Type]
  ------------------------------
  [⊢ (tgt→ τin⁺ τout⁺) ⇒ Type])
```

TURNSTILE+: “Type rules” for types

```
(def-typed-stx (→ τin τout) »
  [⊢ τin » τin+ ⇐ Type]
  [⊢ τout » τout+ ⇐ Type]
  ------------------------------
  [⊢ (tgt→ τin+ τout+) ⇒ Type])

(struct tgt→ (in out))
```

TURNSTILE+: "Type rules" for types

```
(def-typed-stx (→ τin τout) »
  [⊢ τin » τin⁺ ⇐ Type]
  [⊢ τout » τout⁺ ⇐ Type]
  ------------------------------
  [⊢ (tgt→ τin⁺ τout⁺) ⇒ Type])

(struct tgt→ (in out))

(def-pat-stx → …)
```

TURNSTILE+: define-type

```
#lang TURNSTILE+
(define-type → Type Type : Type)
```

```
(def-typed-stx (→ τin τout) »
  [⊢ τin » τin+ ⇐ Type]
  [⊢ τout » τout+ ⇐ Type]
  ------------------------------
  [⊢ (tgt→ τin+ τout+) ⇒ Type])

(struct tgt→ (in out))

(def-pat-stx → …)
```

TURNSTILE+: Binding Types

```
(def-typed-stx (Π [x : τin] τout) »
  [⊢ τin » τin⁺ ⇐ Type]
  [x » x⁺ : τin⁺ ⊢ τout » τout⁺ ⇐ Type]
  ------------------------------
  [⊢ (tgt-Π τin⁺ (tgt-λ x⁺ τout⁺)) ⇒ Type])

(struct tgt-Π (in out))
```

Output must have valid binding structure: allows TURNSTILE+ to handle binding automatically, e.g., `subst`, `type=`, etc

TURNSTILE+: Binding Types

```
#lang TURNSTILE+
(define-type Π #:bind [x : Type] Type : Type)
```

```
(def-typed-stx (Π [x : τ_in] τ_out) »
  [⊢ τ_in » τ_in⁺ ⇐ Type]
  [x » x⁺ : τ_in⁺ ⊢ τ_out » τ_out⁺ ⇐ Type]
  ------------------------------
  [⊢ (tgt-Π τ_in⁺ (tgt-λ x⁺ τ_out⁺)) ⇒ Type])

(struct tgt-Π (in out))
```

Overview

1. Introduce macros and macro-based DSLs
2. Introduce type checking via macros
3. Implement a dependently typed core calculus
4. Scale to a full proof assistant ecosystem

A Dependently Typed Calculus

DEP

```
#lang TURNSTILE+

(define-type Π #:bind [x : Type] Type : Type)

(def-typed-stx (typed-λ [x : τin] e) »
  [⊢ τin » τin⁺ ⇐ Type]
  [x » x⁺ : τin⁺ ⊢ e » e⁺ ⇒ τout⁺]
  --------------------------------
  [⊢ (tgt-λ x⁺ e⁺) ⇒ (Π [x⁺ : τin⁺] τout⁺)]])

(def-typed-stx (typed-app f e) »
  [⊢ f » f⁺ ⇒ (Π [x : τin] τout)]
  [⊢ e » e⁺ ⇐ τin]
  --------------------------------
  [⊢ (β (tgt-app f⁺ e⁺)) ⇒ (subst e⁺ x τout)]])

(define-red β (tgt-app (tgt-λ x e) arg)
              ~> (subst arg x e))
```

"dependent"
=
terms in types

type level computation

TURNSTILE+: Type-level computation, as macros

```
(define-red β (tgt-app (tgt-λ x e) arg)
                ~> (subst arg x e))
```

Contractum = macro output

Redex = macro input

```
(def-stx (β (tgt-app (tgt-λ x e) arg))
   (subst arg x e))
```

Extensible Languages: Type Rules as Libraries

DEP

```
#lang TURNSTILE+

(provide typed-app
         typed-λ)

(def-typed-stx typed-app
   … )

(def-typed-stx typed-λ
   … )
```

Library extends type system with new rule

NAT

```
#lang TURNSTILE+
(provide Z S Nat elim)
; peano nums
(def-type Nat : Type)

(def-type Z : Nat)
(def-type S Nat : Nat)

(def-typed-stx (elim n P mz ms)
   … )
```

PROG1

```
#lang DEP

; uses: typed-app, typed-λ
```

PROG2

```
#lang DEP
(require NAT)
; uses: typed-app, typed-λ,
;       Nat, Z, S, elim-Nat
```

Modular Composable Languages

Reuse entire DEP lang to create new lang

DEP

```
#lang TURNSTILE+

(provide typed-app
         typed-λ)

(def-typed-stx typed-app
   … )

(def-typed-stx typed-λ
   … )
```

PROG1

```
#lang DEP

; uses: typed-app, typed-λ
```

DEP+IND

```
#lang TURNSTILE+
(require+provide dep) ; reuse dep lang
(provide def-datatype)
(def-typed-stx def-datatype
   def-type … ; inductive datatypes
   def-stx …
   def-typed-stx … )
```

PROG2

```
#lang DEP+IND
(require Nat)
; uses: typed-app, typed-λ, def-datatype
; e.g., length-indexed list
(def-datatype Vec : [A : *] : [i : Nat] -> *
  [nil : (Vec A 0)]
  [cons [k : Nat] [x : A] [xs : (Vec A k)]
        : (Vec A (S k))])
```

Core dependently type langs: hard to use

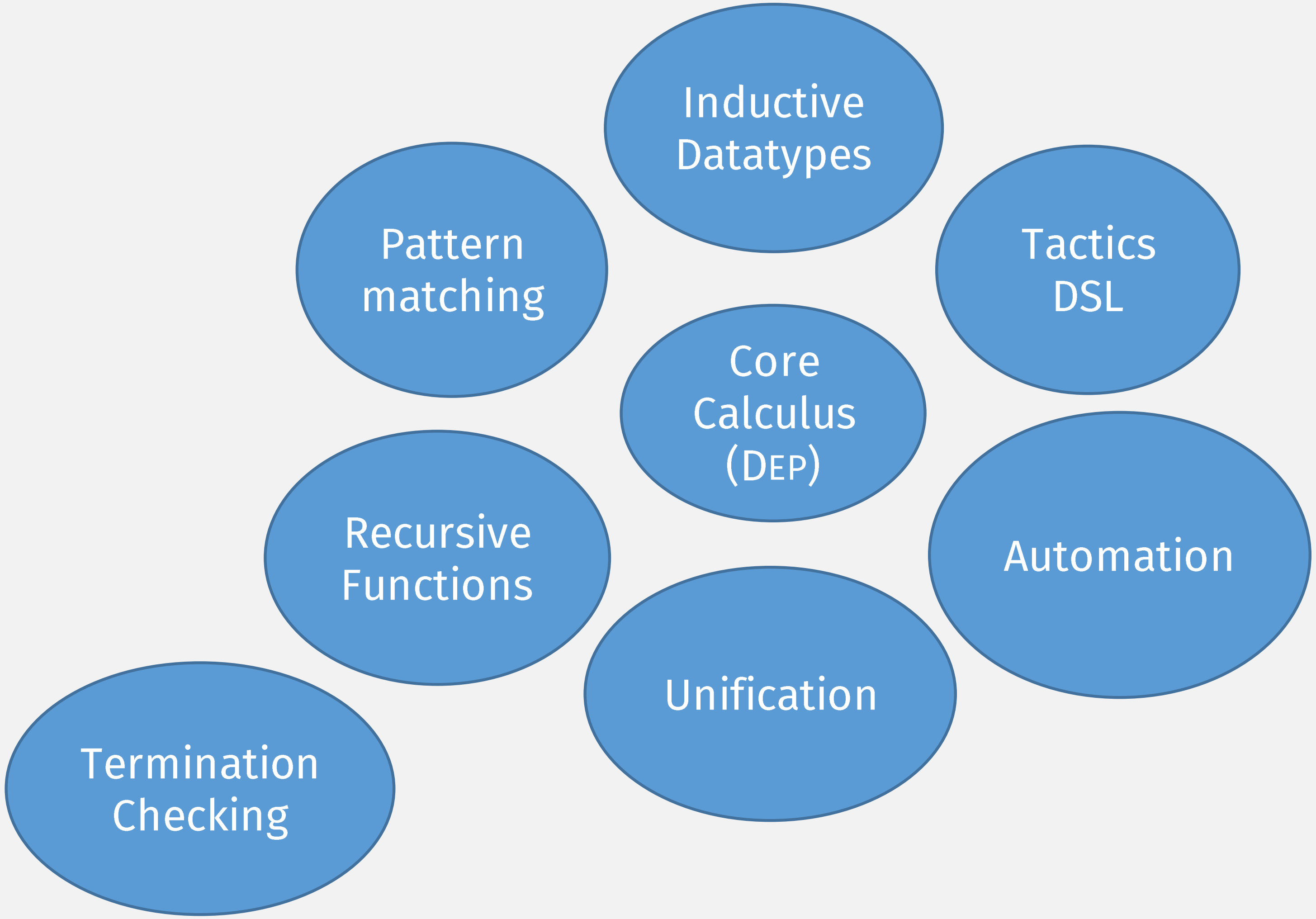
DEP-PROG

```
#lang DEP
; proof of: n + 0 = n
(λ [x : Nat]
   (elim x
    (λ [n : Nat] (= Nat (plus n 0) n))
    (refl Nat 0)
    (λ [x-1 : Nat]
      (λ [ih : (= Nat (plus x-1 0) x-1)]
        (elim
         ih
         ; a=b => a+1=b+1
         (λ [a : Nat] [b : Nat]
            (λ [e : (= Nat a b)]
              (= Nat (s a) (s b))))
         (λ [c : Nat]
           (refl Nat (s c))))))))
 : (Π [x : Nat] (= Nat (plus x 0) x))
```

Overview

1. Introduce macros and macro-based DSLs
2. Introduce type checking via macros
3. Implement a dependently typed core calculus
4. Scale to a full proof assistant ecosystem

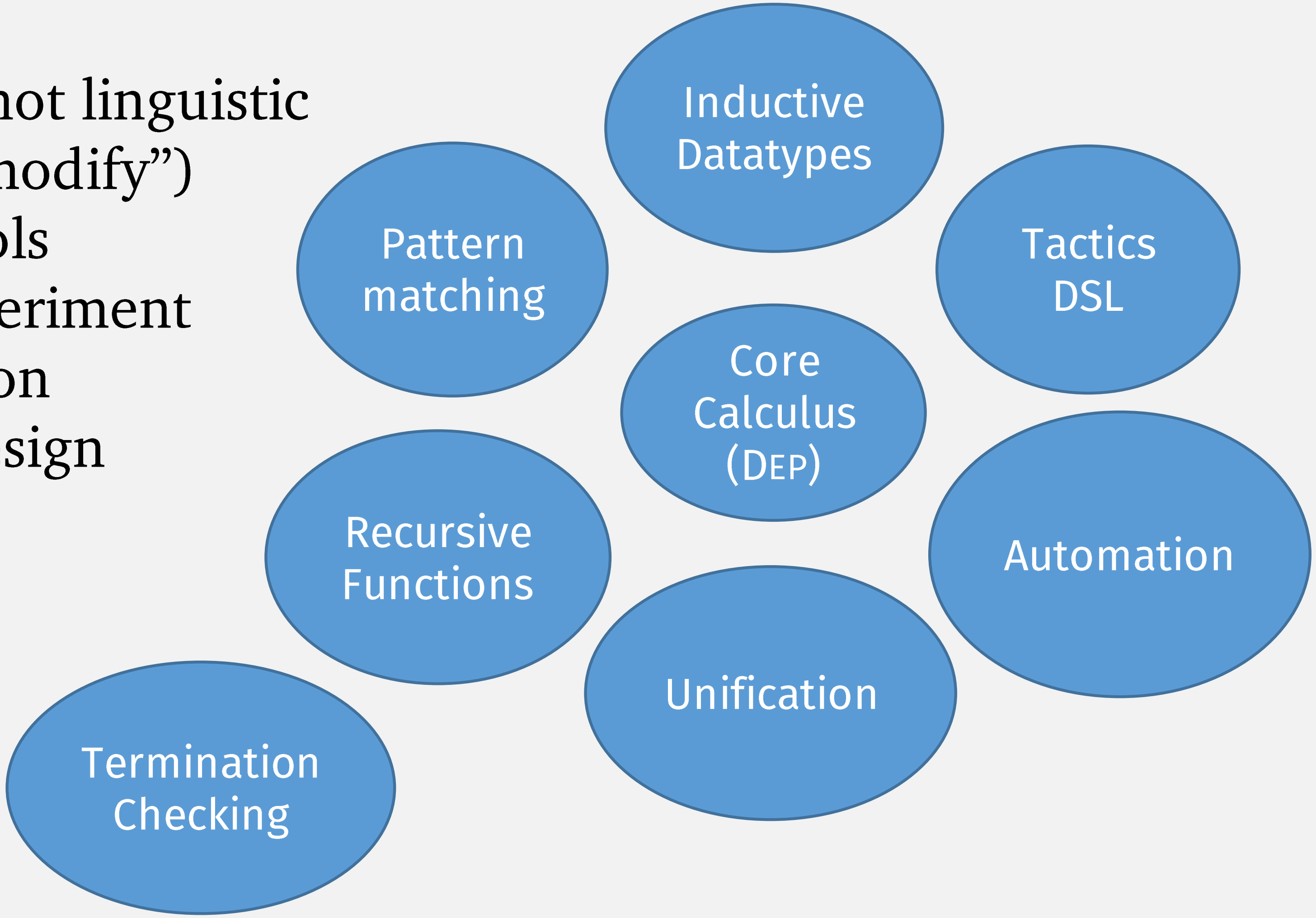
A Proof Assistant is:



A Proof Assistant is:
a Collection of Interacting Extensions and DSLs

Problems:

- Extensions not linguistic ("fork and modify")
- 3rd party tools
- Hard to experiment and iterate on language design



Language-Oriented Programming can Help

The 3 Keys of LOP:

DSL Creation

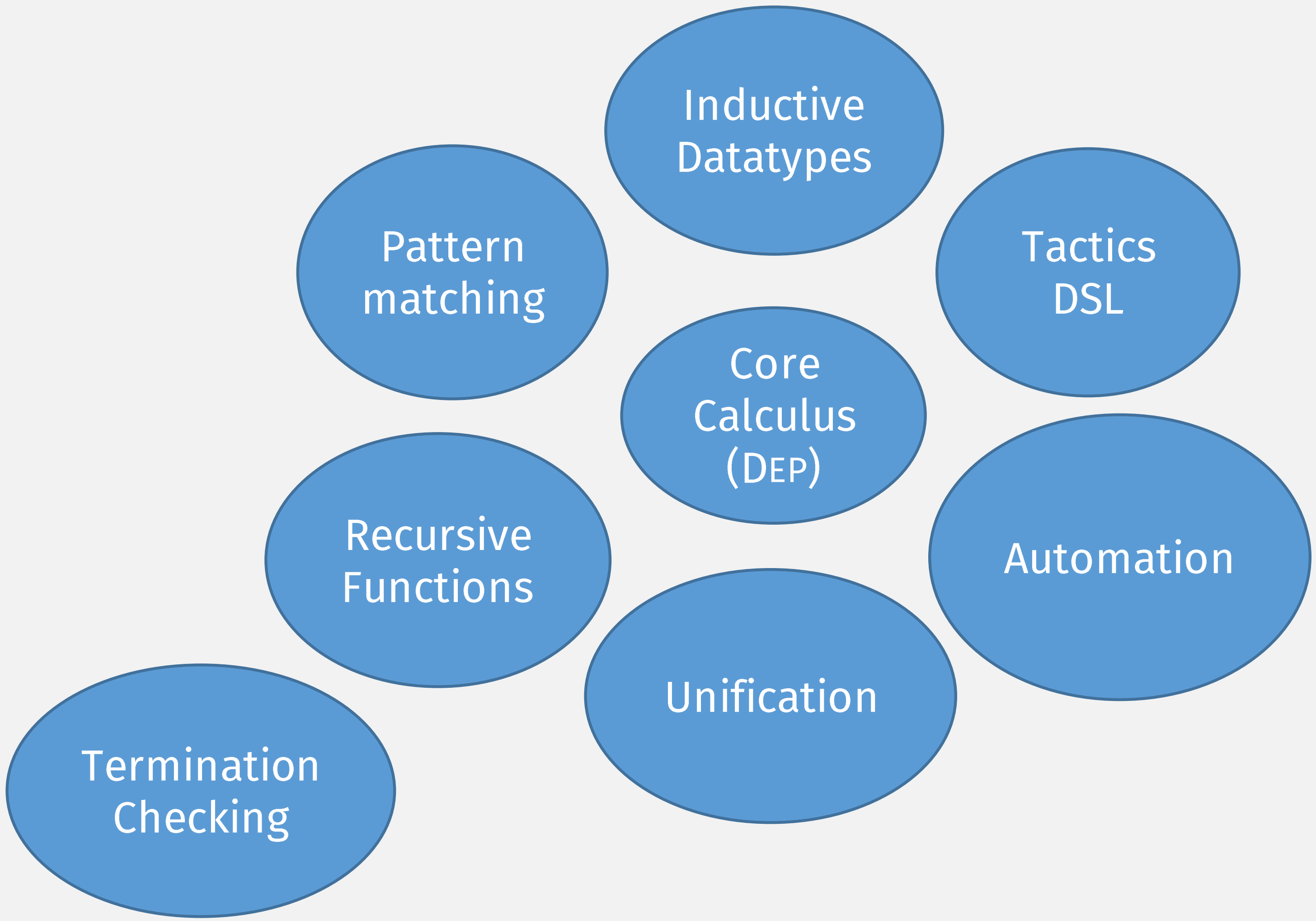
Extension

Integration

"a programming language that enables language-oriented software design (LOP) facilitates:

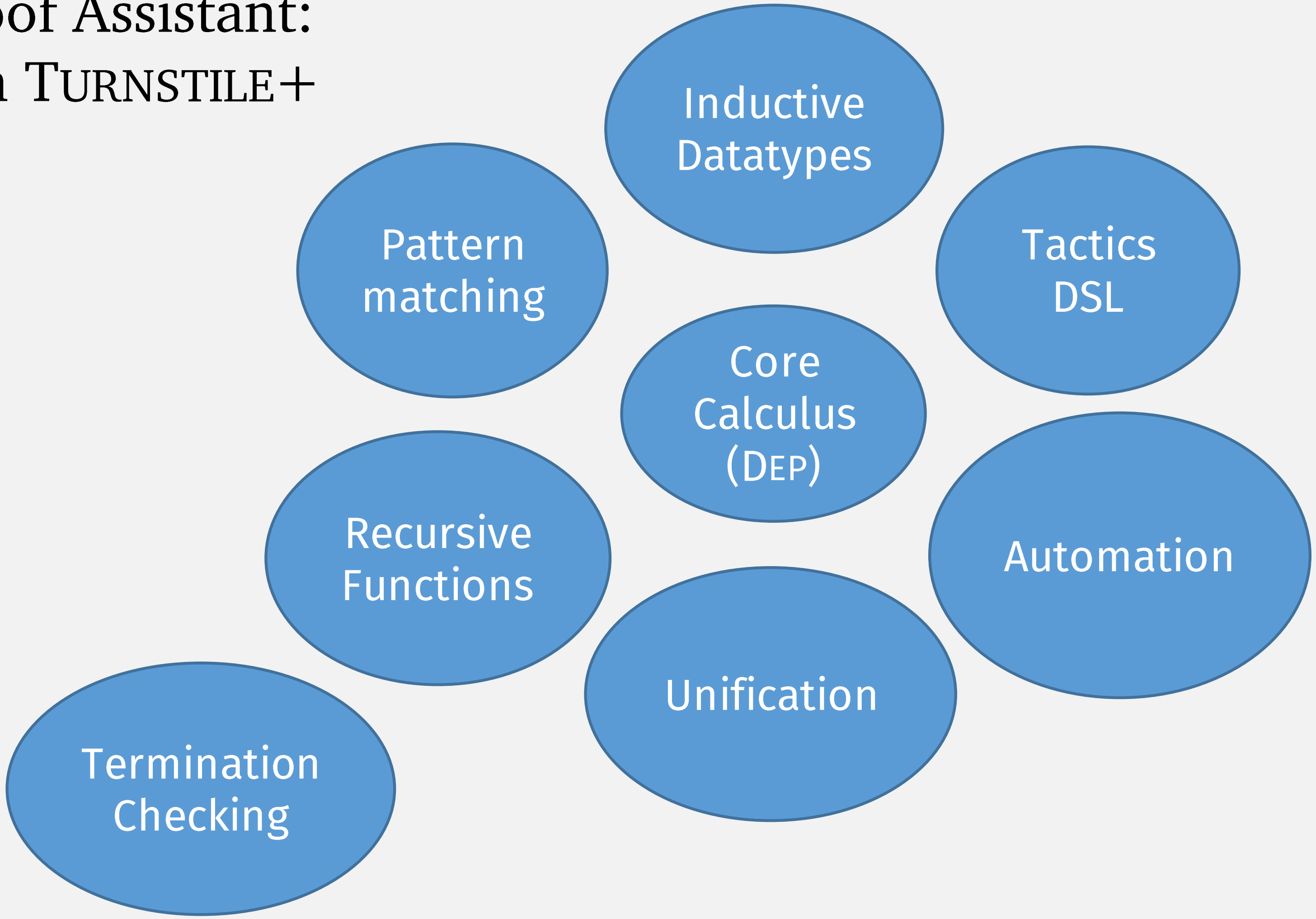
1. easy creation of components as DSLs,
2. Immediate extension of those DSLs, and
3. integration of created and external components."

A Proof Assistant is:
a Collection of Interacting Extensions and DSLs

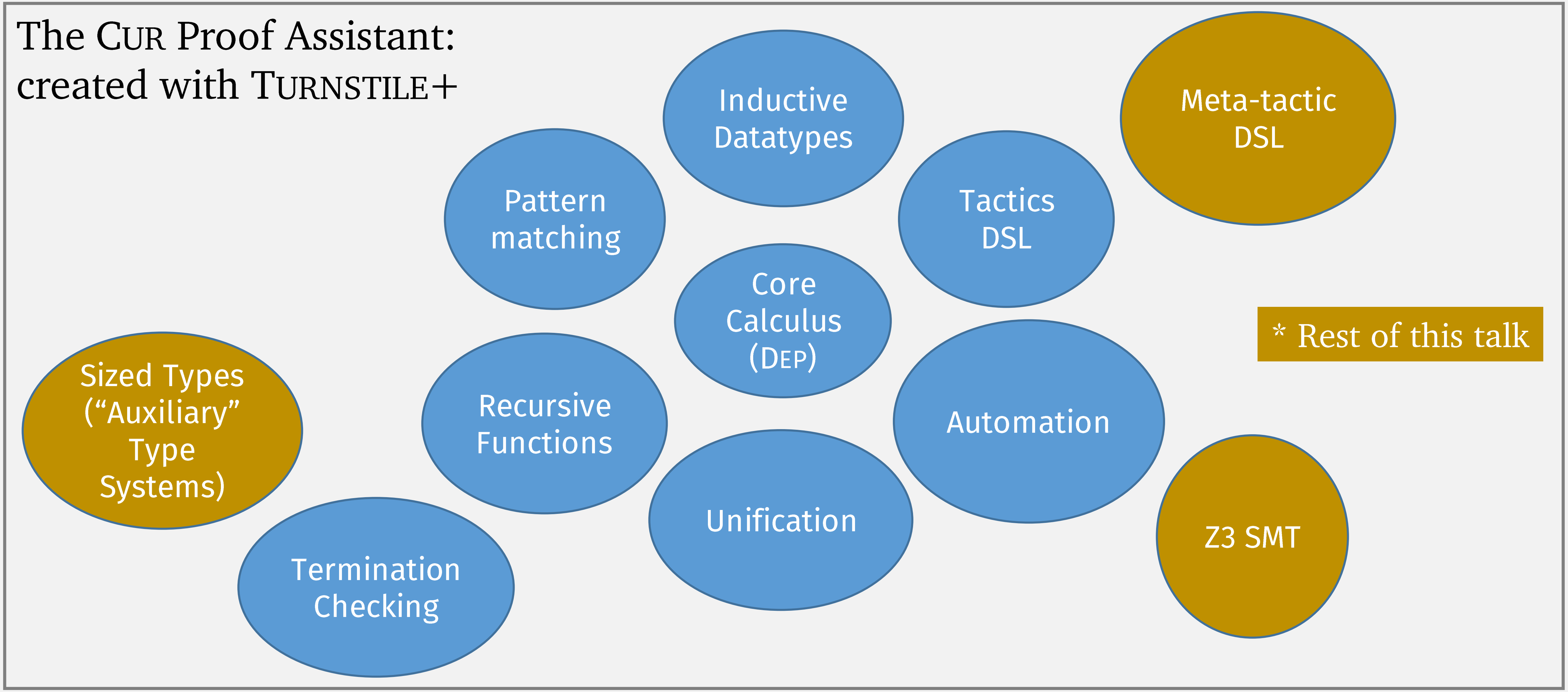


A Proof Assistant is:
a Collection of Interacting Extensions and DSLs

The CUR Proof Assistant:
created with TURNSTILE+



A Proof Assistant is:
a Collection of Interacting Extensions and DSLs



The 3 Keys of LOP Can Help When Creating New Proof Assistant Components

- DSL Creation: a tactic (and metatactic) tactic DSL
- Extension: an experimental library for sized types
- Integration: prototype automation via SMT

The 3 Keys of LOP Can Help When Creating New Proof Assistant Components

- DSL Creation: a tactic (and metatactic) tactic DSL
- Extension: an experimental library for sized types
- Integration: prototype automation via SMT

Interactive Proofs

Ctx	Goal (Type)	Proof (Term)	Tactic
	`Π [X : Type] [x : X] X`	?	

Proof tree data structure

Proof terms can have **holes**

Interactive Proofs

Ctx	Goal (Type)	Proof (Term)	Tactic
	Π [X : Type] [x : X] X	?	
		λX:Type.??	intro X

Ctx	Goal (Type)	Proof (Term)	Tactic
X:Type	Π [x : X] X	??	
X:Type	Π [x : X] X	λx:X.???	intro x

zipper data structure navigates proof tree, points to current focus

Interactive Proofs

Ctx	Goal (Type)	Proof (Term)	Tactic
	Π [X : Type] [x : X] X	?	
		λX:Type.??	intro X

Ctx	Goal (Type)	Proof (Term)	Tactic
X:Type	Π [x : X] X	??	
X:Type	Π [x : X] X	λx:X.???	intro X

current focus

Ctx	Goal (Type)	Proof (Term)	Tactic
X:Type, x:X	X	???	
X:Type, x:X	X	x	assumption

Final proof: λX:Type.λx:X.x

Intro Tactic

NTAC

```
#lang TURNSTILE+
;; usage: (intro X)
(def-stx (intro X:id)
  #'(lambda  (zipnode)
      (define focus (get-focus zipnode))
      (define ctx (get-ctx zipnode))
      (define goal (get-goal focus))
      (define new-prooftree-node
        (match goal
          [(Π (x:id : P:expr) body:expr)
           (make-node
```

A better idea: abstract with a DSL

```
            (make-node/ctx
             (ctx-add ctx X P)
             (make-hole-node (subst X x body))))
           (lambda (proof-of-body)
             (λ (X : P) proof-of-body))]))
      (find-next-goal
       (update-focus zipnode new-prooftree-node))))
```

METANTAC: a DSL for writing ntac tactics

Insert arbitrary holes in proof term

Match on current goal

NTAC

```
#lang METANTAC
(def-tactic (intro X) #:current-goal (Π (x : P) body)
   ($fill (λ (X : P) HOLE1)
         #:where [X : P ⊢ HOLE1 : (subst X x body)]))
```

Update current focus

Holes automatically generate new subgoal(s)

metantac: a DSL for writing ntac tactics

implicit bindings for proof context information

NTAC

```
#lang METANTAC
; ....
(def-tactic assumption
  (unless (for/first ([(k v) $ctxt] #:when (type= v $goal))
            ($fill k)
    (err "could not find matching assumption for " $goal))))
```

No more holes

Iterate over context

The 3 Keys of LOP Can Help When Creating New Proof Assistant Components

- DSL Creation: a tactic (and metatactic) tactic DSL
- Extension: an experimental library for sized types
- Integration: prototype automation via SMT

Termination Checking Recursive Functions

```
#lang cur
(def/match div [n : Nat] [m : Nat] : Nat
  [Z _ => n]
  [(S n-1) m => (S (div (minus n-1 m) m))])
```

Not allowed by standard (syntactic) termination analysis

Sized Types: lift all functions and types

```
sized data SNat : Size -> Set
{ zero : [i : Size] -> SNat ($ i)
; succ : [i : Size] -> SNat i -> SNat ($ i) }

fun minus : [i : Size] -> SNat i -> SNat # -> SNat i
{ minus i (zero (i > j)) y = zero j
; minus i x (zero .#) = x
; minus i (succ (i > j) x) (succ .# y) = minus j x y
}
```

Argument non-increasing

MiniAgda code from [Abel 2010]

Sized Type Disadvantages

- Verbose
- Complicates type checking
  - `minus sz_i x y != minus sz_j x y`

A “Parallel” Type System, with Stx Props

SIZED-LIB

```
(def-stx (lift-datatype TY)
  #:with [C τ] (get-data-constructor #'TY)
  #'(def-stx (C_sz arg)
      (attach-type
        (C arg)
        (attach-size τ (inc-sz arg)))))
```

Given a type, eg **Nat**,
for each constructor, eg **cons**,
define a wrapper, eg **cons**$_{sz}$,
that adds size information to its types

A “Parallel” Type System, with Stx Props

SIZED-LIB

```
(define-tyrule (def/match_sz f [x : τ #:sz i] : τout #:sz j [pat bod] …)
  #:with τ_i   (add-sz #'τ #'i)
  #:with τ_<i (add-sz #'τ #'(< i))
  #:with τout_j   (add-sz #'τout #'j)
  #:with τout_<j (add-sz #'τout #'(< j))
  #:with [x_pat τ_pat] (pat->ctxt #'pat #'τ_i) ; τ_pat … has size (< i)
  [ [x ≫ x⁺ : τ_i]
    [x_pat ≫ x_pat⁺ : τ_pat]
    [f ≫ f⁺ : (Π [x : τ_<i] τout_<j)] ⊢ [bod ≫ bod⁺ ⇐ τout_j] …
    #:where τ= (λ (τ_1 τ_2)
                 (and (τ=_OLD τ_1 τ_2)
                      (sz-ok? (get-sz τ_1) (get-sz τ_2)))) ]
```

Propagate size information

Use size information to type check

Overload type equality (for this rule) to consider sizes

Experimental Sized Types Library in Cur

```
#lang Cur
(require cur/sizedtypes)

(lift-datatype Nat)

(def/match_sz minus_sz [n : Nat #:sz i] [m : Nat] : Nat #:sz i
  [Z_sz _ => n]
  [_ Z_sz => n]
  [(S_sz n-1) (S_sz m-1) => (minus_sz n-1 m-1)])
```

Declare function non-increasing

Experimental Sized Types Library in Cur

```
#lang Cur
(require cur/sizedtypes)

(lift-datatype Nat)

(def/match_sz minus_sz [n : Nat #:sz i] [m : Nat] : Nat #:sz i
  [Z_sz _ => n]
  [_ Z_sz => n]
  [(S_sz n-1) (S_sz m-1) => (minus_sz n-1 m-1)])

(def/match_sz div_sz [n : Nat #:sz i] [m : Nat] : Nat #:sz i
  [Z_sz _ => n]
  [(S_sz n-1) m => (S_sz (div_sz (minus_sz n-1 m) m))])
```

And: `minus  x  y` still equivalent to `minus  x  y` by default

The 3 Keys of LOP Can Help When Creating New Proof Assistant Components

- DSL Creation: a tactic (and metatactic) tactic DSL
- Extension: an experimental library for sized types
- Integration: prototype automation via SMT

Automation via SMT

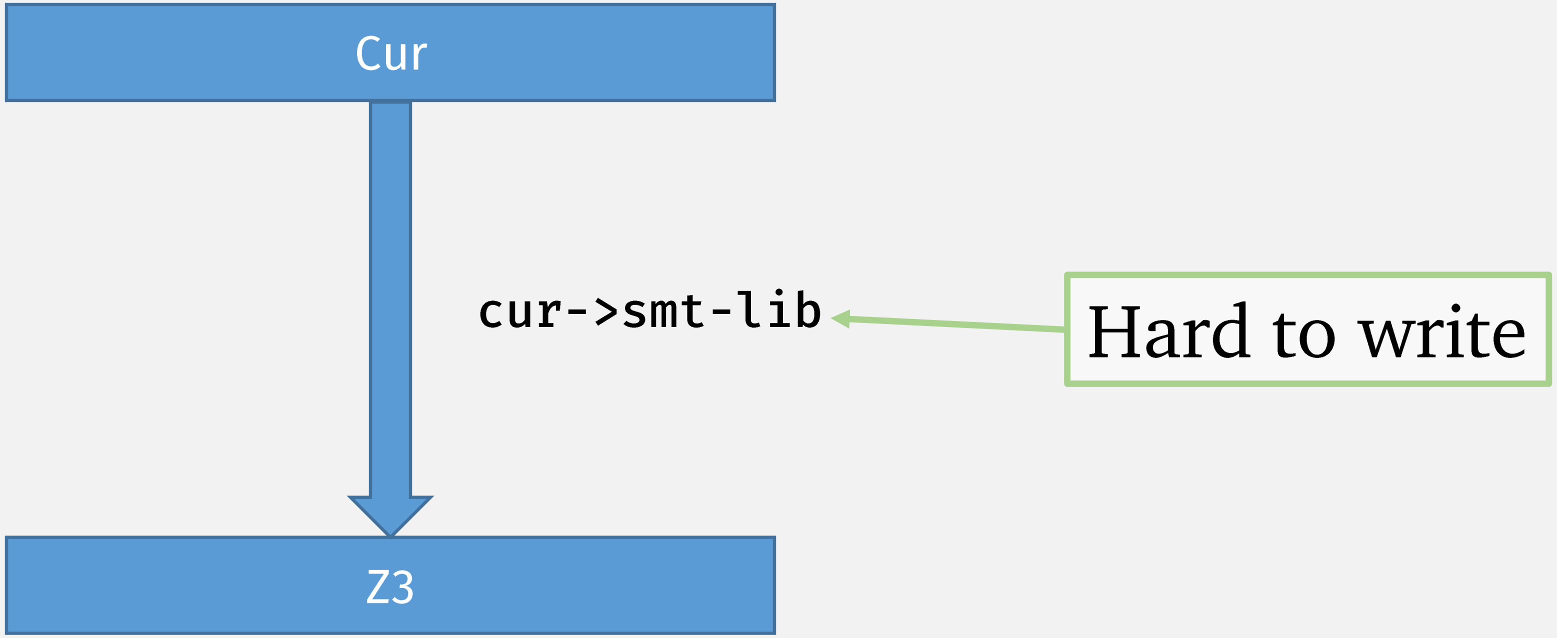
SF-prog

```
#lang cur

(require ntac)

(def-thm minus-diag
  (∀ [n : Nat] (= (minus n n) 0))
  (ntac ?????))
```

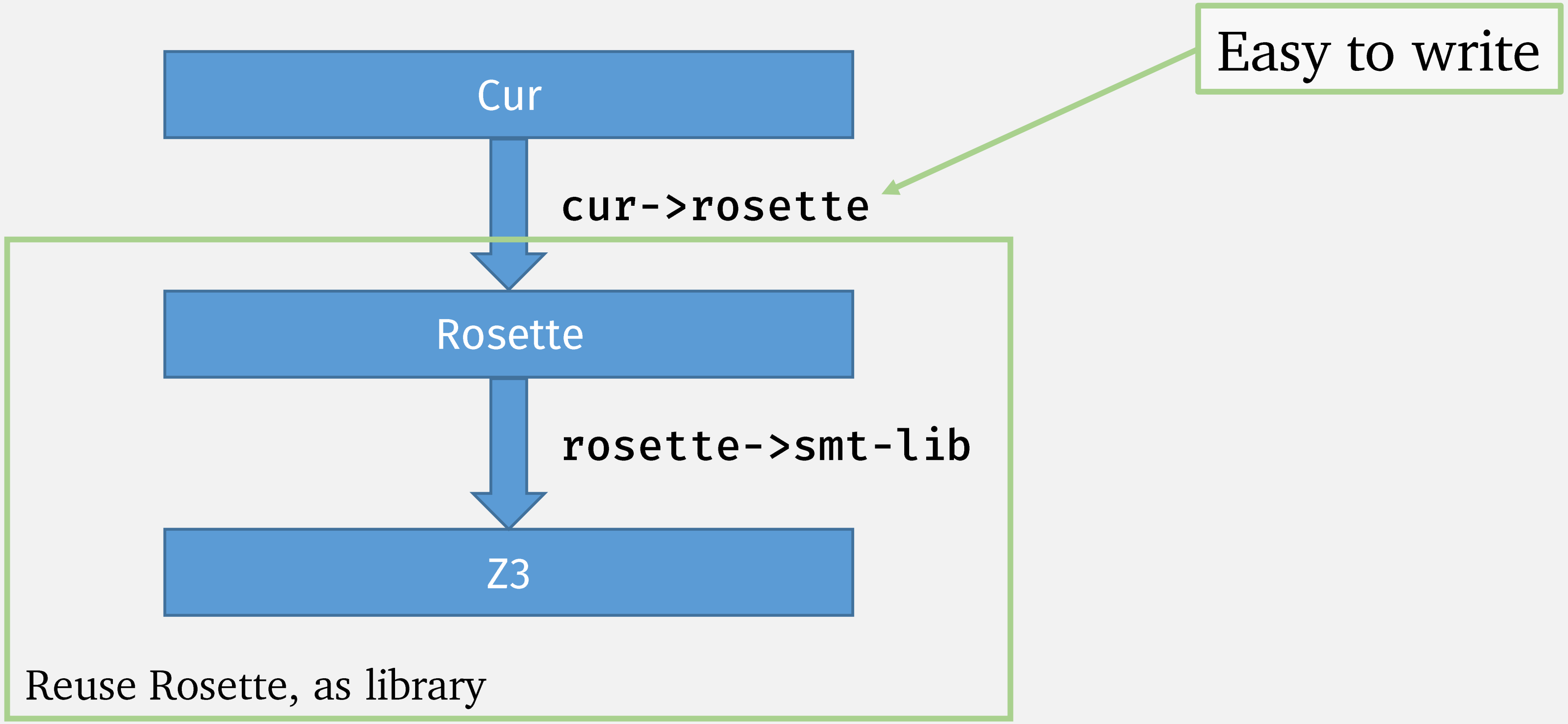
Automation via SMT



The Rosette Language [Torlak and Bodik 14]

“Rosette is a solver-aided programming language that extends Racket with language constructs for program synthesis, verification, and more. To verify or synthesize code, Rosette compiles it to logical constraints solved with off-the-shelf SMT solvers”

Automation via SMT



Automation via SMT

Z3-AXIOM-LIB

```
(def-stx (def-z3-axiom name τ)
  (def rosette-result (rosette:verify (cur->rosette #'τ))
  #:fail-unless (unsat? rosette-result)
                "Rosette could not prove:” #’τ
                “counterexample:” rosette-result
  #’(def-typed-stx name
       (attach (attach #’fresh-name ‘z3-axiom) #’τ)))
```

Create term with desired type

Automation via SMT

Z3-AXIOM-LIB

```
(def-stx (def-z3-axiom name τ)
  (def rosette-result (rosette:verify (cur->rosette #'τ))
  #:fail-unless (unsat? rosette-result)
                "Rosette could not prove:” #’τ
                “counterexample:” rosette-result
  #’(def-typed-stx name
       (attach (attach #’fresh-name ‘z3-axiom) #’τ)))
```

Create term with desired type

But label as axiom

Automation via SMT

Z3-AXIOM-LIB

```
(def-stx (def-z3-axiom name τ)
  (def rosette-result (rosette:verify (cur->rosette #'τ))
  #:fail-unless (unsat? rosette-result)
                "Rosette could not prove:” #’τ
                “counterexample:” rosette-result
  #’(def-typed-stx name
       (attach (attach #’fresh-name ‘z3-axiom) #’τ)))


(def-stx (print-z3-axioms e)
  (find #’e ‘z3-axiom))
```

Create term with desired type

But label as axiom

Find “z3-axiom” tags

Automation via SMT

SF-prog

```
#lang cur

(require ntac)

(def-thm minus-diag
  (∀ [n : Nat] (= (minus n n) 0))
  (ntac ?????))
```

Automation via SMT

SF-prog

```
#lang cur

(require z3-axiom-lib)

(define-z3-axiom minus-diag
  (∀ [n : Nat] (= (minus n n) 0)))
```

Automation via SMT

SF-prog

```
#lang cur

(require z3-axiom-lib)

(define-z3-axiom minus-diag
  (∀ [n : Nat] (= (minus n n) 0)))

(print-z3-axioms minus-diag)
; axioms used by minus-diag:
; - minus-diag: (∀ [n : Nat] (= (minus n n) 0)))
```

Automation via SMT

SF-prog

```
#lang cur

(require z3-axiom-lib)

(define-z3-axiom minus-diag
  (∀ [n : Nat] (= (minus n n) 0)))

(print-z3-axioms minus-diag)
; axioms used by minus-diag:
; - minus-diag: (∀ [n : Nat] (= (minus n n) 0)))

(def-z3-axiom eq-refl
  (∀ [a b : Nat] (-> (= a a) (= a b))))
; => Rosette could not prove (∀ [a b : Nat] (-> (= a a) (= a b)))),
;    counterexample: a = 0, b = 1
```

Takeaways

- TURNSTILE+, via macros , enables easy creation of typed languages
- The 3 keys of LOP:
  - Easy DSL creation,
  - Extension, and
  - Integration

  help with design and experimentation

  in systems of components, like proof assistants.

https://github.com/stchang/macrotypes/tree/cur
https://github.com/wilbowma/cur/tree/turnstile-core