Enriching Typed Racket with Dependent Types

Overview and Status Report

Andrew M. Kent and Sam Tobin-Hochstadt

Presented at PL Wonks @ Indiana University, 17 Apr 2015

The Big Picture

The Big Picture

- Typed Racket

The Big Picture

- Typed Racket + refinement types

The Big Picture

- Typed Racket + refinement types + linear integer constraints

The Big Picture

- Typed Racket + refinement types + linear integer constraints
- *Q: Haven't those things been done before?*

The Big Picture

- Typed Racket + refinement types + linear integer constraints
- *Q: Haven't those things been done before?*
- Individually? Absolutely!

The Big Picture

- Typed Racket + refinement types + linear integer constraints
- *Q: Haven't those things been done before?*
- Individually? Absolutely!
  - **sound interoperation** with full featured dynamically typed language

The Big Picture

- Typed Racket + refinement types + linear integer constraints
- *Q: Haven't those things been done before?*
- Individually? Absolutely!
  - **sound interoperation** with full featured dynamically typed language
  - **unique type system** successfully used by Typed Racket and Typed Clojure

The Big Picture

- Typed Racket + refinement types + linear integer constraints
- *Q: Haven't those things been done before?*
- Individually? Absolutely!
  - **sound interoperation** with full featured dynamically typed language
  - **unique type system** successfully used by Typed Racket and Typed Clojure
  - no dependence on SMT solver

The Big Picture

- Typed Racket + refinement types + linear integer constraints
- *Q: Haven't those things been done before?*
- Individually? Absolutely!
  - **sound interoperation** with full featured dynamically typed language
  - **unique type system** successfully used by Typed Racket and Typed Clojure
  - no dependence on SMT solver *(more traditional type system)*

Already Logical Types

A Brief Refresher on How Typed Racket Works!

Already Logical Types

- Question: How do we type a Lisp-like language?

Already Logical Types

- Question: How do we type a Lisp-like language?
- Simple example:

Already Logical Types

- Question: How do we type a Lisp-like language?
- Simple example:
  - `(+ 1 2)`

Already Logical Types

- Question: How do we type a Lisp-like language?
- Simple example:
  - `(+ 1 2)`
  - `+` is of type `Number -> Number -> Number`

Already Logical Types

- Question: How do we type a Lisp-like language?
- Simple example:
  - `(+ 1 2)`
  - `+` is of type `Number -> Number -> Number`
  - `1` and `2` are of type `Number`

Already Logical Types

- Question: How do we type a Lisp-like language?
- Simple example:
  - `(+ 1 2)`
  - `+` is of type `Number -> Number -> Number`
  - `1` and `2` are of type `Number`
  - therefore `(+ 1 2)` is of type `Number`

Already Logical Types

- Question: How do we type a Lisp-like language?
- Less simple example:

Already Logical Types

- Question: How do we type a Lisp-like language?
- Less simple example:
  - ```
    (define (plus1 n)
      (if (fixnum? n)
          (fx+ n 1)
          (fl+ n 1.0)))
    ```
    ```

Already Logical Types

- Question: How do we type a Lisp-like language?
- Less simple example:
  - ```
    (define (plus1 n)
      (if (fixnum? n)
          (fx+ n 1)
          (fl+ n 1.0)))
    ```
  - `n` is *either* a `Fixnum` or a `Flonum`
    ```

Already Logical Types

- Question: How do we type a Lisp-like language?
- Less simple example:
  - ```
    (define (plus1 n)
      (if (fixnum? n)
          (fx+ n 1)
          (fl+ n 1.0)))
    ```
  - `n` is *either* a `Fixnum` or a `Flonum`
  - What does `(fixnum? n)` tell us?
    ```

Already Logical Types

- Question: How do we type a Lisp-like language?
- Less simple example:
  - ```
    (define (plus1 n)
      (if (fixnum? n)
          (fx+ n 1)
          (fl+ n 1.0)))
    ```
  - `n` is *either* a `Fixnum` or a `Flonum`
  - What does `(fixnum? n)` tell us?
- Answer: We need **logical propositions** about types!
    ```

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- Assume `(n -: Fixnum)` *or* `(n -: Float)`

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- Assume `(n -: (U Fixnum Float))`

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- Assume `(n -: (U Fixnum Float))`
- **then branch**...?

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- Assume `(n -: (U Fixnum Float))`
- **then branch** `(n -: Fixnum)`

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- Assume `(n -: (U Fixnum Float))`
- **then branch** `(n -: Fixnum)`
  - `(fx+ n 1)` typechecks!

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- Assume `(n -: (U Fixnum Float))`
- **then branch** `(n -: Fixnum)`
  - `(fx+ n 1)` typechecks!
- **else branch**...?

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- Assume `(n -: (U Fixnum Float))`
- **then branch** `(n -: Fixnum)`
  - `(fx+ n 1)` typechecks!
- **else branch** `(n -! Fixnum)`

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- Assume `(n -: (U Fixnum Float))`
- **then branch** `(n -: Fixnum)`
  - `(fx+ n 1)` typechecks!
- **else branch** `(n -! Fixnum)`

  ⇒ `(n -: Float)`

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- Assume `(n -: (U Fixnum Float))`
- **then branch** `(n -: Fixnum)`
    - `(fx+ n 1)` typechecks!
- **else branch** `(n -! Fixnum)`

  ⇒ `(n -: Float)`
    - `(fl+ n 1.0)` typechecks!

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- Simple types won't cut it!

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- Simple types won't cut it!
  - `(Γ ⊢ (fixnum? n)`
    `: Boolean)`

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- We need logical types!

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- We need logical types!

  - ```
    (Γ ⊢ (fixnum? n)
       : Boolean
       ; when not #f => (n -: Fixnum)
       ; when #f     => (n -! Fixnum)
       ; ∅)
    ```
    ```

Already Logical Types

- Typechecking our example:

```
(define (plus1 n)
  (if (fixnum? n)
      (fx+ n 1)
      (fl+ n 1.0)))
```

- We need logical types!

  ○
  ```
  (Γ ⊢ (fixnum? n)
     : Boolean
     ; when not #f => (n -: Fixnum)
     ; when #f     => (n -! Fixnum)
     ; ∅)
  ```

- Typed Racket uses logical propositions as part of the typing judgement and in function types

More Descriptive Types

So what is the type of plus1?

More Descriptive Types

So what is the type of plus1?

(Hint... we're going to leverage those logical propositions!)

More Descriptive Types

- The type of **`plus1`**:

More Descriptive Types

- The type of **`plus1`**:
- Good

```
[(U Fixnum Float) -> (U Fixnum Float)]
```

More Descriptive Types

- The type of `plus1`:
- Good

```
[(U Fixnum Float) -> (U Fixnum Float)]
```

- But we'd like to know `(plus1 1)` produces a `Fixnum`

More Descriptive Types

- The type of `plus1`:
- Better

```
(case-> [Fixnum -> Fixnum]
        [Float -> Float]
        [(U Fixnum Float)
         -> (U Fixnum Float)])
```

More Descriptive Types

- The type of `plus1`:
- Better

```
(case-> [Fixnum -> Fixnum]
        [Float -> Float]
        [(U Fixnum Float)
         -> (U Fixnum Float)])
```

- But we don't want to forget the relation between input and output types!

More Descriptive Types

- The type of `plus1`:

- e.g. we want this to typecheck:

```
(define (foo [x : (U Fixnum Float)]) : Fixnum
  (let ([y (plus1 x)])
    (if (fixnum? x)
        (fx* x y)
        42)))
```

More Descriptive Types

- The type of `plus1`:

- e.g. we want this to typecheck:

```
(define (foo [x : (U Fixnum Float)]) : Fixnum
  (let ([y (plus1 x)])
    (if (fixnum? x)
        (fx* x y)
        42)))
```

- current type of `plus1`:

```
(case-> [Fixnum -> Fixnum]
        [Float -> Float]
        [(U Fixnum Float)
         -> (U Fixnum Float)])
```

More Descriptive Types

- The type of `plus1`:
- Best

```
(dependent-case->
 [Fixnum -> Fixnum]
 [Float -> Float])
```

More Descriptive Types

- The type of `plus1`:

- Best

```
(dependent-case->
 [Fixnum -> Fixnum]
 [Float -> Float])
```

- Now this typechecks!

```
(define (foo [x : (U Fixnum Float)]) : Fixnum
  (let ([y (plus1 x)])
    (if (fixnum? x)
        (fx* x y)
        42)))
```

More Descriptive Types

- The type of `plus1`:
- Best

```
(dependent-case->
 [Fixnum -> Fixnum]
 [Float -> Float])
```

- ... but how *is* `dependent-case->` implemented?

More Descriptive Types

- The type of `plus1`:
- Best

```
(dependent-case->
 [Fixnum -> Fixnum]
 [Float -> Float])
```

- expands to:

```
[([x : (U Fixnum Float)])
 ->
 (Refine [ret : (U Fixnum Float)]
         (or (and [x   -: Fixnum]
                  [ret -: Fixnum])
             (and [x   -: Float]
                  [ret -: Float])))]
```

More Descriptive Types

- The type of `plus1`:
- Best

```
(dependent-case->
 [Fixnum -> Fixnum]
 [Float -> Float])
```

- refinement types = more precise types!

```
[([x : (U Fixnum Float)])
 ->
 (Refine [ret : (U Fixnum Float)]
         (or (and [x   -: Fixnum]
                  [ret -: Fixnum])
             (and [x   -: Float]
                  [ret -: Float])))]
```

More Descriptive Types

```
[([x :  (U Fixnum Float)])
 ->
 (Refine [ret : (U Fixnum Float)]
         (or (and [x   -: Fixnum]
                  [ret -: Fixnum])
             (and [x   -: Float]
                  [ret -: Float])))]
```

- Sound interoperability?

More Descriptive Types

```
[([x :  (U Fixnum Float)])
 ->
 (Refine [ret : (U Fixnum Float)]
         (or (and [x   -: Fixnum]
                  [ret -: Fixnum])
             (and [x   -: Float]
                  [ret -: Float])))]
```

- Sound interoperability?
  - with typed code?

More Descriptive Types

```
[([x :  (U Fixnum Float)])
 ->
 (Refine [ret : (U Fixnum Float)]
         (or (and [x   -: Fixnum]
                  [ret -: Fixnum])
             (and [x   -: Float]
                  [ret -: Float])))]
```

- Sound interoperability?
    - with typed code? Typechecker!

More Descriptive Types

```
[([x : (U Fixnum Float)])
 ->
 (Refine [ret : (U Fixnum Float)]
         (or (and [x   -: Fixnum]
                  [ret -: Fixnum])
             (and [x   -: Float]
                  [ret -: Float])))]
```

- Sound interoperability?
  - with typed code? Typechecker!
  - with untyped code?

More Descriptive Types

```
[([x : (U Fixnum Float)])
 ->
 (Refine [ret : (U Fixnum Float)]
         (or (and [x   -: Fixnum]
                  [ret -: Fixnum])
             (and [x   -: Float]
                  [ret -: Float])))]
```

- Sound interoperability?
  - with typed code? Typechecker!
  - with untyped code? Contracts!

More Descriptive Types

```
[([x : (U Fixnum Float)])
 ->
 (Refine [ret : (U Fixnum Float)]
         (or (and [x   -: Fixnum]
                  [ret -: Fixnum])
             (and [x   -: Float]
                  [ret -: Float]))))]
```

- Sound interoperability?

```
(->i ([x (or/c fixnum? flonum?)])
     [ret (x) (if (fixnum? x) fixnum? flonum?)])
```

More Descriptive Types

- Another example

More Descriptive Types

- #racket: prevent flonum divide-by-zero error

More Descriptive Types

- #racket: prevent flonum divide-by-zero error
- Assume `(denom -: Float)` and `(ε -: Positive-Float)`

More Descriptive Types

- #racket: prevent flonum divide-by-zero error
- Assume `(denom -: Float)` and `(ε -: Positive-Float)`

```
(cond
  [(fl> (flabs denom) ε)
   <division-exp>]
  ...)
```

More Descriptive Types

- #racket: prevent flonum divide-by-zero error
- Assume `(denom -: Float)` and `(ε -: Positive-Float)`

```
(cond
  [(fl> (flabs denom) ε)
   <division-exp>]
  ...)
```

- In `<division-exp>` we know `denom ≠ 0`, but currently in TR this fact is lost

More Descriptive Types

- #racket: prevent flonum divide-by-zero error
- Assume `(denom -: Float)` and `(ε -: Positive-Float)`

```
(cond
  [(fl> (flabs denom) ε)
   <division-exp>]
  ...)
```

- In `<division-exp>` we know `denom ≠ 0`, but currently in TR this fact is lost
- `(flabs denom)` types at `Fixnum` currently

More Descriptive Types

- #racket: prevent flonum divide-by-zero error
- Assume `(denom -: Float)` and `(ε -: Positive-Float)`

```
(cond
  [(fl> (flabs denom) ε)
   <division-exp>]
  ...)
```

- In `<division-exp>` we know `denom ≠ 0`, but currently in TR this fact is lost
- With a dependent refinement we could more accurately track these types!

More Descriptive Types

- #racket: prevent flonum divide-by-zero error
- Assume `(denom -: Float)` and `(ε -: Positive-Float)`

```
(cond
  [(fl> (flabs denom) ε)
   <division-exp>]
  ...)
```

- In `<division-exp>` we know `denom ≠ 0`, but currently in TR this fact is lost
- With a dependent refinement we could more accurately track these types!
  - Just like `plus1` using `dependent-case->` to better track types

Linear Integer Constraints

Relating more than just 'types'!

Linear Integer Constraints

- Types can depend on other types...

Linear Integer Constraints

- Types can depend on other types...
- What about other practical, decidable theories?

Linear Integer Constraints

- Types can depend on other types...
- What about other practical, decidable theories?
- **Linear integer constraints** are a well understood, decidable problem w/ numerous applications!

Linear Integer Constraints

```
(define (norm [v : (Vectorof Real)])
  (sqrt (for/sum ([i (vec-len v)])
          (square (vector-ref v i)))))
```

Linear Integer Constraints

```
(define (norm [v : (Vectorof Real)])
  (sqrt (for/sum ([i (vec-len v)])
          (square (vector-ref v i)))))
```

- Is it possible to get an out of bounds error from `(vector-ref v i)`?

Linear Integer Constraints

```
(define (norm [v : (Vectorof Real)])
  (sqrt (for/sum ([i (vec-len v)])
          (square (vector-ref v i)))))
```

- Is it possible to get an out of bounds error from `(vector-ref v i)`?
- Nope!
  `∀i (and (≤ 0 i) (< i (vec-len v)))`

Linear Integer Constraints

```
(define (norm [v : (Vectorof Real)])
  (sqrt (for/sum ([i (vec-len v)])
          (square (vector-ref v i)))))
```

- The optimizer can replace **`vector-ref`** with **`unsafe-vector-ref`**:

Linear Integer Constraints

```
(define (norm [v : (Vectorof Real)])
  (sqrt (for/sum ([i (vec-len v)])
          (square (vector-ref v i)))))
```

- The optimizer can replace **`vector-ref`** with **`unsafe-vector-ref`**:
- This requires no intervention from the user!

Linear Integer Constraints

```
(: safe-vector-ref
   (All (α) (([v : (Vectorof α)]
              [i : Natural (< i (vec-len v))])
             -> α)))
(define safe-vector-ref vector-ref)
```

Linear Integer Constraints

```
(: safe-vector-ref
   (All (α) (([v : (Vectorof α)]
              [i : Natural (< i (vec-len v))])
             -> α)))
(define safe-vector-ref vector-ref)
```

- Users can specifically require statically guaranteed safe usages of functions like `vector-ref`

Linear Integer Constraints

```
(: safe-vector-ref
   (All (α) (([v : (Vectorof α)]
              [i : Natural (< i (vec-len v))])
             -> α)))
(define safe-vector-ref vector-ref)
```

- Users can specifically require statically guaranteed safe usages of functions like `vector-ref`
- `safe-vector-ref` can never have a runtime out-of-bounds error!

Linear Integer Constraints

```
(define (dot-product [v1 : (Vectorof Real)]
                     [v2 : (Vectorof Real)
                         (= (vec-len v1)
                            (vec-len v2))])
  (for/sum ([i (vec-len v1)])
    (* (safe-vector-ref v1 i)
       (safe-vector-ref v2 i))))
```

Linear Integer Constraints

```
(define (dot-product [v1 : (Vectorof Real)]
                     [v2 : (Vectorof Real)
                         (= (vec-len v1)
                            (vec-len v2))])
  (for/sum ([i (vec-len v1)])
    (* (safe-vector-ref v1 i)
       (safe-vector-ref v2 i))))
```

- No bounds errors + verified optimizations!

Linear Integer Constraints

- What about **`plus1`**?

Linear Integer Constraints

- What about `plus1`?

```
(dependent-case->
 [[n : Fixnum] -> [m : Fixnum
                     (= m (+ 1 n))]]
 [Float -> Float])
```

Recap

- Refinements are a great, natural extension to Typed Racket!

Recap

- Refinements are a great, natural extension to Typed Racket!
  - Relate the types of runtime values and reason about common integer constraints!

Recap

- Refinements are a great, natural extension to Typed Racket!
  - Relate the types of runtime values and reason about common integer constraints!
  - Create expressive dependent functions!

Recap

- Refinements are a great, natural extension to Typed Racket!
  - Relate the types of runtime values and reason about common integer constraints!
  - Create expressive dependent functions!
  - The logical propositions are already part of the type system!

Recap

- Refinements are a great, natural extension to Typed Racket!
    - Relate the types of runtime values and reason about common integer constraints!
    - Create expressive dependent functions!
    - The logical propositions are already part of the type system!
    - Refinements are easily mapped to dependent contracts

Recap

- Refinements are a great, natural extension to Typed Racket!
  - Relate the types of runtime values and reason about common integer constraints!
  - Create expressive dependent functions!
  - The logical propositions are already part of the type system!
  - Refinements are easily mapped to dependent contracts
  - Small/medium scale implemented and working! (Currently scaling to full scale...)

To Do

Next Steps?

To Do

- Finish implementing these features

To Do

- Finish implementing these features
- Experiment w/ new types for standard library

To Do

- Finish implementing these features
- Experiment w/ new types for standard library
- Support arbitrary *pure* predicates in refinements

To Do

- Finish implementing these features
- Experiment w/ new types for standard library
- Support arbitrary *pure* predicates in refinements

Thanks!

- PLT Redex model available: **`https://github.com/andmkent/stop2015-redex`**
  - Typed Racket fork: **`https://github.com/andmkent/typed-racket`**
    - Work in progress! =)