Adding Practical Dependent Types to Typed Racket

Andrew M. Kent
Indiana University
andmkent@indiana.edu

Sam Tobin-Hochstadt
Indiana University
samth@indiana.edu

1. Introduction

Typed Racket is a statically-typed dialect of Racket that allows idiomatic Racket programs to be enriched with types. It can reason about many dynamically typed programming patterns *while* providing sound interoperability and optimizations. We have designed and are implementing an extension to Typed Racket which adds support for logical refinement types and linear integer constraints. This summary discusses our approach to implementing this novel combination of precise specifications and optimizations while maintaining sound interoperability with dynamically typed code. A PLT Redex model of the basic calculus describing our extension can be seen at `https://github.com/andmkent/stop2015-redex` and our development fork where we are extending Typed Racket is available at `https://github.com/andmkent/typed-racket`.

2. Refining *already* logical types

In order to typecheck dynamically typed idioms Typed Racket utilizes *occurrence typing*, a technique which allows different occurrences of the same term in a program to be typechecked at different types. This is accomplished by tracking the type-based logical information implied by the results of conditional tests. For example, to typecheck the function `plus1` we begin with the logical assumption that `x` is of type `(U Fixnum Float)` (*an untagged union type describing values of type* `Fixnum` *or* `Float`) :

```
(define (plus1 [x : (U Fixnum Float)])
  (if (fixnum? x)
      (fx+ x 1)
      (fl+ x 1.0)))
```

We then typecheck the body of the function starting with the `if`'s test-expression, `(fixnum? x)`, recording the type-based logical propositions its result would imply: if non-`#f` then `x` *is* a `Fixnum`, otherwise `x` *is not* a `Fixnum`. To typecheck the then- and else-branches, we combine the respective implied proposition with our initial assumption that `x` is of type `(U Fixnum Float)`. This allows us to correctly conclude that `x` is a `Fixnum` in `(fx+ x 1)` and a `Float` in `(fl+ x 1.0)`.

Our extension utilizes these same logical typed-based propositions *within types* to naturally extend Typed Racket to relate the types of different values. To this end, we have added **logical refinement types** of the form $\{x : \tau \mid \psi\}$. This type defines a subset of type τ where the logical proposition ψ holds, allowing us to more precisely type many programs. For example, consider what a precise type for `plus1` might look like without these refinements:

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

The type constructor `case->` builds an ordered function intersection type, which serves as a simple overloading technique that can express how prior knowledge about the argument types gives us a more specific return type. Unfortunately, the relation between input and output type is easily lost: calling `plus1` with something of type `(U Fixnum Float)`, for example, means the fact that the return value is of type `Fixnum` iff the argument was of type `Fixnum` is forgotten. The typechecker simply chooses the first valid function type based on the information available at the application site.

By using a simple logical refinement, however, we can exactly describe the relation between the function's argument and return type:

```
([in : (U Fixnum Float)]
 ->
 [out : (U Fixnum Float)
      (or (and [in  : Fixnum]
               [out : Fixnum])
          (and [in  : Float]
               [out : Float]))])
```

This type refines the function's range using a logical proposition, stating the type of the argument (`in`) and result (`out`) must agree. Since this is a common pattern, we can use a spoonful of syntactic-sugar to help users describe these sorts of dependent functions without concerning themselves with the exact details of the logical propositions:

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

This allows for a convenient balance between precise specification and readable, intuitive type definitions and better models this common pattern found in dynamically typed programs.

3. Verifying integer constraints

In addition to supporting refinements that relate run-time values' types, we support a decidable subset of **integer constraints** similar to those presented by Xi and Pfenning (1998) in Dependent ML. With this addition, Typed Racket will be able to automatically verify and eliminate many runtime checks for numeric constraints. For example, in the program `norm` below, Typed Racket will be able to safely replace `vector-ref` with its faster counterpart `unsafe-vector-ref` since the bounds-safety requirement can be statically guaranteed:

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

Furthermore, our extension will allow users to explicitly require the static enforcement of integer constraints by including them in the refinements of types. This allows a user to benefit from

otherwise risky manual optimizations—such as explicit uses of `unsafe-vector-ref`—in a safe, statically verified fashion:

```
(define (safe-vec-ref
         [v : (Vectorof Real)]
         [i : Natural (< i (vec-len v))])
  (unsafe-vector-ref v i))
```

This extension also provides additional ways a program's specification may be gradually converted into static types. Here we can see how the typechecker can enforce the precondition for vector `dot-product` by explicitly requiring the passed vectors be of equal length:

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

Similarly, by applying these refinements to the return types of functions like `plus1` we can more accurately describe their behavior so their usages can be more precisely checked:

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

4. Interoperating with the dynamically typed world

Because of the relatively simple nature of our dependent types, it is easy to see the mapping between our type extensions and the well studied dependent contracts already present in Racket (Dimoulas et al. 2012). For example, the dependent specification we gave for `plus1` is easily expressed as a run-time contract:

```
(->i ([in (or/c fixnum? flonum?)])
     [out (in) (or/c (and/c fixnum?
                            (=/c (+ 1 in)))
                     flonum?)])
```

Because of this, we can provide sound, performant interoperability between traditional Racket modules and Typed Racket modules utilizing dependent types with no additional effort from our users.

5. Dependently typing the numeric tower

Racket, like many dynamically typed languages in the Lisp tradition, features a rich numeric tower. The developers of Typed Racket have taken great care to ensure the types assigned to operations involving this tower are appropriately expressive (St-Amour et al. 2012). This means that seemingly simple operations, such as `+`, may actually have a quite specific (*and thus verbose*) type to allow more programs to typecheck:

```
(case->
 [Pos-Byte Pos-Byte -> Pos-Index]
 [Byte Byte -> Index]
 [Index Pos-Index -> Pos-Fixnum]
 [Pos-Index Index Index ->  Pos-Fixnum]
 ...
 [Number * -> Number])
```

Because of the overlap between types in the numeric tower and refined integers, we plan to explore the benefits and costs of using integer refinements in place of some of these simpler nominal integer subtypes (e.g. `Byte`). It seems certain that more programs will typecheck when utilizing the more detailed types integer refinements can describe, but it is unclear how this will impact performance when typechecking large and complex numeric libraries. Upon completing our implementation of this system we plan to explore and report on these costs.

6. Related Works

There is a history of using refinements and dependent types to enrich already existing type systems. Our method borrows inspiration from Xi and Pfenning (1998), who added a practical set of dependent types to ML to allow for richer specifications and compiler optimizations. We similarly strive to provide an expressive yet practical extension which preserves the decidability of typechecking and requires no external support (*e.g. an SMT solver*). Our approach, however, is designed explicitly to reason about a dynamically typed programming language and provide sound interoperability with code that provides no static type-guarantees.

Chugh et al. (2012b) have explored how extensive use of dependent refinement types and an SMT solver can enable typechecking for rich dynamically type languages such as JavaScript (Chugh et al. 2012a). Our system's usage of refinements to express dynamically typed idioms is similar, but we use a different approach for typechecking programs: our system is designed to allow for the direct expression of the mutually dependent subtyping and proves relations instead of translating problems into SMT-formatted queries. This allows us to express and reason about subtyping in the more traditional way and prevents potentially expensive external logical queries when relatively simple subtyping checks will suffice. Additionally, since our system is more tightly integrated with the compilation process, we can provide optimizations and guarantees of sound interoperability that a system such as Dependent JavaScript cannot.

Sage's use of dynamic runtime checks and static types is similar to our approach for providing sound interoperability between dynamically and statically typed values, however their usage of first-class types and arbitrary refinements means typechecking is undecidable (Knowles et al. 2007). Our system instead utilizes a more conservative, decidable approach and forgos the use of types as first-class objects to maintain consistency between typed and untyped Racket programs. Also, our system is not limited to reasoning about a purely functional language.

Bibliography

Ravi Chugh, David Herman, and Ranjit Jhala. Dependent Types for JavaScript. In *Proc. ACM Conf. Object-Oriented Programming, Systems, Languages and Applications*, pp. 587–606, 2012a.

Ravi Chugh, Patrick Maxim Rondon, and Ranjit Jhala. Nested Refinements: A Logic for Duck Typing. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 231–244, 2012b.

Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete Monitors for Behavioral Contracts. In *Proc. European Sym. on Programming*, pp. 214–233, 2012.

Kenneth Knowles, Aaron Tomb, Jessica Gronski, Stephen N. Freund, and Cormac Flanagan. Sage: Unified Hybrid Checking for First-Class Types, General Refinement Types, and Dynamic (Extended Report). 2007.

Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the Numeric Tower. In *Proc. Sym. on Practical Aspects of Declarative Languages*, pp. 289–303, 2012.

Hongwei Xi and Frank Pfenning. Eliminating Array Bound Checking Through Dependent Types. In *Proc. ACM Conf. Programming Language Design and Implementation*, pp. 249–257, 1998.