Typing the Numeric Tower

Vincent St-Amour[1], Sam Tobin-Hochstadt[1], Matthew Flatt[2], and Matthias Felleisen[1]

[1] Northeastern University
{stamourv,samth,matthias}@ccs.neu.edu
[2] University of Utah
mflatt@cs.utah.edu

Abstract. In the past, the creators of numerical programs had to choose between simple expression of mathematical formulas and static type checking. While the Lisp family and its dynamically typed relatives support the straightforward expression via a rich numeric tower, existing statically typed languages force programmers to pollute textbook formulas with explicit coercions or unwieldy notation. In this paper, we demonstrate how the type system of Typed Racket accommodates both a textbook programming style and expressive static checking. The type system provides a hierarchy of numeric types that can be freely mixed as well as precise specifications of sign, representation, and range information—all while supporting generic operations. In addition, the type system provides information to the compiler so that it can perform standard numeric optimizations.

1 Designing the Numeric Tower

From the classic two-line factorial program to financial applications to scientific computation to graphics software, programs rely on numbers and numeric computations. Because of this spectrum of numeric applications, programmers wish to use a wide variety of numbers: the inductively defined natural numbers, fixed-width integers, floating-point numbers, complex numbers, etc. Supporting this variety demands careful attention to the design of programming languages that manipulate numbers.

Most languages have taken one of two approaches to numbers. Many untyped languages, drawing on the tradition of Lisp and Smalltalk, provide a hierarchy of numbers whose various levels can be freely used together, known as the *numeric tower*. For example, the following Racket expression mixes arbitrary precision integers with inexact floating-point numbers and produces a complex result:

```
(sqrt (/ 3.14159 (- (expt 2 32))))
```

That is, the numeric tower supports concise expression of mathematical formulas.

Other languages provide static checking of various numeric operations, ensuring that results conform to machine representations of numbers. Static checking helps programmers reason about the requirements, behavior, and performance of their programs. Some languages also provide a limited ability to combine different forms of numbers together in arithmetic operations for a small set of numeric representations. No existing typed language provides as rich a numeric hierarchy nor as many generic operations as those found in Smalltalk, Scheme, or Racket.

In this paper, we describe the design of the numeric tower in Typed Racket (Tobin-Hochstadt and Felleisen 2008), which combines expressiveness with static checking. Typed Racket is an explicitly and statically typed sister language to Racket, a mostly-functional language (Flatt and PLT 2010). Using Typed Racket, programmers may convert untyped Racket programs by adding explicit type declarations. In the existing type system, we can encode fine distinctions in the hierarchy of numeric types and express numerous mathematical properties of numeric operations in their types. Combining these features allows programmers to state and enforce static properties about their numeric programs while maintaining the concise mathematical expression of untyped Racket. Furthermore, we can reuse standard optimization techniques to reap the performance benefits of static typing.

Three features of Typed Racket support this design. Due to *true union types* (Buneman and Pierce 1999) the choices of numeric types do not need to reflect the underlying runtime representation of numbers nor do they affect the representation. For example, `Integer` is the union of positive and negative integers, yet Racket's run-time representation has no knowledge of this division. Due to *overloading with intersection types* (Coppo and Dezani-Ciancaglini 1978; Reynolds 1988) the type system supports precise specification of the behavior of numeric operations such as `+` without necessitating multiple implementations. Thus it can express that adding two positive integers produces a positive integer and adding a negative integer to a negative floating point number produces a negative floating point value. Due to *occurrence typing* (Tobin-Hochstadt and Felleisen 2008, 2010) the type checker can "lower" the numeric types of variables based on dynamic tests including predicates and numeric comparisons.

The remainder of the paper begins with a series of examples that illustrate Typed Racket's approach to numeric computations. We then describe the encoding of the type hierarchy in section 3, the typing of numeric operations using overloading in section 4, and the use of occurrence typing to refine types in section 5. Finally, in section 6, we describe our implementation, focusing on challenges concerning usability.

2 A Rich Numeric Tower

We introduce Typed Racket and its approach to numeric programming with a series of small examples. The mathematical absolute value function, $|-|$, takes real numbers to non-negative real numbers. As figure 1 shows, a programmer can naturally express this simple fact via types. Furthermore, the function definition itself transliterates the textbook definition of `abs` into the concrete syntax of a functional programming language; Typed Racket's type system accomplishes the rest.

The `pythagorean` function also benefits from encoding sign information in the type system. Racket's `sqrt` function, like its mathematical counterpart, optionally may yield complex numbers. Programmers often write programs, however, that depend on real-valued results from `sqrt`. To accommodate the latter, the type of `sqrt` in Typed Racket maps non-negative reals to non-negative reals. Because the square of any real number is provably always non-negative and the sum of two non-negative numbers is also non-negative, the type system can validate that the length of the hypotenuse of any right triangle is non-negative. See the type of the `pythagorean` function in figure 1.

```
(: abs : Real → Nonnegative-Real)
(define (abs x) (if (> x 0) x (- x)))
```

```
(: pythagorean : Real Real → Nonnegative-Real)
(define (pythagorean a b) (sqrt (+ (sqr a) (sqr b))))
```

```
(: nat->hex : Natural → (Listof Byte))
(define (nat->hex n)
  (cond [(= n 0) '()]
        [else (cons (modulo n 16) (nat->hex (quotient n 16)))]))
```

```
(: sum-vector : (Vectorof Integer) → Integer)
(define (sum-vector v)
  (define n (vector-length v))
  (let loop ([i 0] [sum 0])
    (if (< i n) (loop (+ i 1) (+ sum (vector-ref v i))) sum)))
```

```
(: gen-random : Float Float → Float)
(define (gen-random min max)
  (next) (+ min (/ (* (- max min) x) p)))
(define p (- (expt 2 31) 1))
(define A (expt 7 5))
(define x 42) ; state of the PRNG
(define (next) (set! x (modulo (* A x) p))) ; x_{i+1} ≡ A · x_i (mod p)
```

Figure 1: Numeric programs in Typed Racket

Sign properties are a special case of range properties, another common set of properties that programmers want to establish. For instance, two program fragments may need to communicate via a protocol that limits the range of encoded values. A type system that supports subtyping and overloading makes it possible to mix and match fixed-width and unbounded integers, both widely used by Racket programmers. Hence, programmers can have the mathematically correct behavior of unbounded integers as the default and may still enforce range properties when desired, without explicit coercions.

In the third example of figure 1, Typed Racket's type system guarantees that the result of `(modulo n 16)` fits within a byte. Arguments to `nat->hex` can be unbounded integers, and the range properties still hold. Similarly, in the fourth example, the type system guarantees that `i` is of type `Index`, which is bounded by the maximum length of Racket vectors. This ensures that vector index computations are performed directly using machine arithmetic instead of costly arbitrary precision operations; all index computations in `sum-vector` use machine integers directly. Furthermore, the results of these functions can be freely mixed with unbounded integers in subsequent computations, without introducing explicit coercions.

The ability to freely mix numbers from different levels of the numeric tower in arithmetic expressions is another convenience of blackboard mathematics that is important for programmers. Again, many type systems require explicit coercions for mixed value expressions, as in Standard ML, Ocaml or Haskell, or provide a limited set of built-in implicit coercions, as in C or Java. A type system that can encode the promotion rules of arithmetic operations when used on operands of mixed types saves the programmer from having to repeatedly encode these rules in his programs in an ad-hoc manner.

The last example in figure 1 presents an implementation of Lewis et al. (1969)'s multiplicative congruential pseudo-random number generator that features mixed-type arithmetic with integer and floating-point numbers. Local type inference (Pierce and Turner 2000) determines that `p`, `A` and `x` are of type `Integer` and both arguments to `gen-random` (`min` and `max`) are of type `Float`. In the boldface section, the result of the subtraction, a floating-point number, is multiplied by an integer, which results in a floating-point number. This implementation is structurally that of a textbook; the actual mathematical operations are unobscured by coercions or other artifacts.

3 Encoding the Numeric Hierarchy

To encode arithmetic specifications, a type system must classify numbers. For example, if a type system is to encode specifications involving sign properties, it needs to distinguish between positive and negative numbers at the type level. To reason about exactness of results, a type system needs to encode exactness of numbers as part of their type. We express distinctions along these axes using true unions and subtyping.

3.1 Union types

Typed Racket provides general union types. For example, `(U Integer Float)` contains all integers as well as all floating-point numbers. Subtyping follows the usual rules for union types, e.g., both the `Integer` type and `Float` type are subtypes of `(U Integer Float)`. It thus is possible to use either an integer or a floating-point number as a value of the union without injection.

Since true unions do not add tags to the values of their constituents, they do not impose constraints on the underlying machine-level representation of data, which has several benefits. First, we can overlay a type hierarchy on top of Racket's existing representations, without requiring changes to the compiler and runtime. Second, we can make finer-grained distinctions at the type level than at the representation level. For example, while Racket uses the same IEEE 754 floating-point representation scheme for positive and negative floating-point numbers, Typed Racket distinguishes the two at the type level by providing both a `Positive-Float` and a `Negative-Float` type. Finally, because we build our numeric types as unions of non-overlapping base types, the intersection of any two numeric types is necessarily a union of some of those non-overlapping base types and thus denotes a valid type. Therefore, we reap some of the benefits of useful intersection types without the need for general intersection types.

3.2 Layers of Numbers

Figure 2 shows Typed Racket's *layers* of the numeric hierarchy. Most layers correspond to well-known sets, such as integers, rationals, and complex numbers. Others correspond to numbers with specific machine representations, such as floating-point.

These layers are similar to the numeric types offered by most programming languages. In addition to the usual integer and floating-point layers, Typed Racket offers exact rationals and both exact and floating-point complex numbers. Members of these layers are integrated with the rest of the numeric tower: operations on numbers from other layers of the tower can produce rationals or complex numbers. For example, the result of dividing 2 by 5 is the fraction $\frac{2}{5}$. Rationals and complex numbers can also be mixed freely with numbers from the other layers of the tower; e.g., the addition of the rational $\frac{2}{5}$ and the integer 3 yields the rational $\frac{17}{5}$.

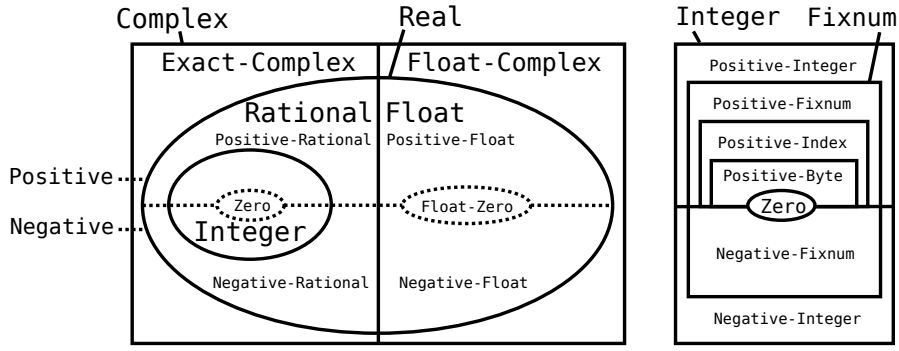

Figure 2: Typed Racket's numeric type hierarchy, with signs

Layers are related by subtyping in the expected fashion; `Integer` is a subtype of `Rational`, which is a subtype of `Real`. `Float` is also a subtype of `Real`. All numeric types in Typed Racket are subtypes of `Complex`. For convenience, Typed Racket provides a `Number` type as a synonym for `Complex`; we use the two interchangeably.

3.3 Signed Types

As a refinement of layers, Typed Racket distinguishes numbers based on their sign. Typed Racket offers positive and negative subdivisions of all layers except `Complex`, as well as types for integer zero and for both of the floating-point zeroes, producing types such as `Positive-Integer` and `Negative-Rational`.

In addition, unions can express types such as `Natural`, which corresponds to the union of `Positive-Integer` and `Zero`, the type of the integer zero. It would be possible to subdivide the `Complex` layer into quadrants to achieve a similar effect.

The different zero types are singleton types that contain only the appropriate zero. Singleton types for first-order values are straightforward, and fit nicely in our subtyping hierarchy. For convenience, a `Float-Zero` type containing both floating-point zeroes is provided, as well as a `Real-Zero` type that also includes the integer zero. Since these different zero values cause distinct behavior when used as arguments, we distinguish them both at the type level as well as at the value level. As explained in section 5, zero types are most useful for comparisons. Since comparisons are not defined on complex numbers, complex zero types are of limited usefulness and are not provided.

As expected, the sign distinctions preserve the subtyping of layers: `Positive-Integer` is a subtype of `Positive-Rational`. In addition, signed subsets are subtypes of their parent layer: `Positive-Integer` is also a subtype of `Integer`. In fact, `Integer` is the union of `Positive-Integer`, `Negative-Integer` and `Zero`. Similarly, the `Positive` type is the union of the "positive" types. Figure 2 shows how sign distinctions refine the numeric hierarchy; see the dotted lines.

3.4 Encoding Range Information

The integer layer is further subdivided into fixed-width integer types, corresponding to different ranges. The `Byte` type contains the integers from 0 to 255, the `Index` type is bounded by zero and the length of the longest possible Racket vector. The `Fixnum` type contains all integers that Racket stores as tagged machine integers on every platform.[3] Those ranges capture a large number of use cases in existing Racket programs. Other ranges could be provided as needed. To prevent base types from overlapping, we fix the ranges when implementing the type system. In addition, bounds on range types are static; bounds cannot depend on values, unlike in dependently-typed systems.

Sign distinctions can also apply to these types to express types such as `Positive-Byte` or `Negative-Fixnum`. These types are also related by subtyping: `Byte` is a subtype of `Index`, which is a subtype of `Nonnegative-Fixnum`. Figure 2 shows a close-up view of the subdivisions of the integer layer.

4 Typing Operations

To exploit our numeric tower, we need type signatures for primitive operations that are generic and yet as tightly specified as possible. For example, if `x` and `y` are `Integer`s, then `(+ x y)` is also an `Integer`, yet if `x` and `y` are `Float-Complex` numbers, then that should be the result type as well. In this section, we present both the properties of our basic type environment and the mechanisms for expressing such properties.

4.1 Overloading with Ordered Intersection Types

Untyped Racket already provides overloading for numeric operations. The function `+` produces exact results when given exact inputs, and otherwise produces inexact results. The challenge for Typed Racket is to represent these overloadings in the type system and to refine them using the distinctions that only exist in the type environment.

[3] This last point is discussed further in section 6.4

We use *ordered intersection types* to express the multiple possible behaviors of numeric operations. Intersection types (Coppo and Dezani-Ciancaglini 1978; Reynolds 1988) are well known in the type system literature and have been extensively studied in many contexts. In their most general form, intersection types are too expressive and undecidable. Typed Racket instead offers a pragmatic flavor of intersections of function types.[4] To increase the predictability of the system for programmers, intersections of function types are considered in order, with earlier types taking precedence over later types.[5] Thus, the following types are equivalent in Typed Racket:

$$\top \to \top \qquad (\top \to \top) \wedge (\texttt{Integer} \to \texttt{Integer})$$

because $\top$ is a supertype of `Integer`, and thus the first conjunct applies in all possible cases. Typed Racket provides the `case`$\to$ type constructor to build function intersection types. Using this type constructor, we can express this fragment of the type of `+`:

```
(: + (case→ (Integer Integer → Integer)
            (Float   Float   → Float)
            (Number  Number  → Number)))
```

Although intersection types are useful for functions that branch based on their input type, conditionals are not required. For example, this program type checks:

```
(: f (case→ (Number → Number) (String → Number))
(define (f x) 0))
```

A function definition with an intersection type must check properly for *each* branch in the intersection, but no other restrictions are imposed.

In the remainder of this section, we consider several varieties of numeric operations, and show how fine-grained numeric types and overloading via ordered intersections help us express a variety of semantic properties in our type system.

4.2 Simple Numeric Operations

The most basic use of overloading for numeric operations is to express the closure properties of arithmetic operations such as `+` and $*$. For example, the type of `+` includes conjuncts specifying that the sum of two `Integer`s is an `Integer`, and that the sum of two `Real` numbers is also `Real`. These properties hold for $*$ as well.

Signed types provide scope for expressing useful mathematical properties. For example, the type fragment

```
(Negative-Real Negative-Real → Positive-Real)
```

[4] Intersections of function types are especially interesting because they reify overloading.

[5] The need for ordered intersection types is further motivated by Racket's `case-lambda` construct whose operational behavior calls for an ordered execution of clauses.

expresses that $*$ produces positive real numbers when given negative inputs. Using overloading, the types of `*`, `/`, and other operations can express these properties precisely.

Range-bounded types are trickier, because they enjoy fewer closure properties. For example, the sum of two `Index`es may not be an `Index` itself, because integer addition in Racket can exceed the length of the largest possible vector. However, the sum of two `Index`es *is* a `Fixnum`.

Further, Racket and Typed Racket support mixed-typed arithmetic. Hence, the types of primitive operations must describe this behavior as well. For example, the sum of a `Float` and an `Integer` is a `Float` by the promotion rules for addition.

4.3 Other Operations

Many numeric operations in Racket have semantic properties that are expressible using the combination of overloading and our numeric type hierarchy. For example, the `modulo` operation, when given a bounded modulus, produces a bounded result. This property is key to typing the `nat->hex` function in figure 1. Other operations that are given similarly expressive types include `floor`, `ceiling`, and `round`.

Coercion operators can also be precisely typed using overloading. The function `exact->inexact`, which converts exact integers and rationals to floating point values, comes with a type that includes the clauses `Real` $\rightarrow$ `Float` and `Complex` $\rightarrow$ `Float-Complex`.

Finally, some operations have special properties on parts of their domain. The `sqrt` function potentially produces `Complex` results, but it produces `Nonnegative-Real`s for non-negative inputs. The `pythagorean` function in figure 1 relies on this overloading to prove that the third side of a right triangle always has non-negative length.

5 Refining Types with Dynamic Tests

In many cases programs use dynamic tests to determine properties of numeric values. The Typed Racket type system uses occurrence typing (Tobin-Hochstadt and Felleisen 2008, 2010) to refine the types in the program using dynamic type tests such as `exact-integer?` and `positive?`. In addition, we can also express useful properties of comparison operators using occurrence typing, thus refining types even further.

5.1 Numeric Predicates

The key idea of occurrence typing is expressed with the `abs` function from figure 1:

```
(: abs : Real → Nonnegative-Real)
(define (abs x) (if (positive? x) x (- x)))
```

To check this function, the type checker proceeds as follows.

- The function signature assigns `Real` to `x`.
- Based on `(positive? x)`, the type system determines that if the condition holds, `x` must have the type `Positive` in the *then* branch.

– Since restricting `Real` to `Positive` produces a subtype of `Nonnegative-Real`, the *then* branch type checks correctly.
– In the *else* branch, the type of `x` must be both a `Real` and not `Positive`, yielding the `Nonpositive-Real` type, which via negation yields the `Nonnegative-Real` type, as desired.

To express that `positive?` determines whether its argument has type `Positive`, its type is

$$\texttt{positive?} : \left(x : \texttt{Real} \xrightarrow{\texttt{Positive}_x} \texttt{Boolean}\right)$$

The annotation above the arrow is a *proposition* about the parameter x. Specifically, it says that x is positive if the result is true and it is not positive otherwise. With types such as this one, Typed Racket understands many more numeric predicates than simply `positive?`, including `real?`, `inexact?`, `fixnum?` and others.

Because Typed Racket has a precise type hierarchy, a wide variety of predicates can refine types. For example, `Positive` is a union of positive integers represented both as machine integers and bignums, as well as positive exact rationals and positive floating point numbers. A type hierarchy with coarser-grained distinctions would sacrifice some of the precision available for describing the behavior of `positive?`.

Another advantage of occurrence typing in combination with numeric predicates is that it greatly reduces the need for explicit downward coercions within the numeric hierarchy. For example, the following function verifies that its input is an exact integer:

```
(: assert-exact-integer : Any → Integer)
(define (assert-exact-integer in)
  (if (exact-integer? input) in (error "not an integer")))
```

Without occurrence typing, this program would require an explicit injection into the exact integer type. Instead, we leverage both the untagged union representation of Racket numbers and the handling of predicates by the type system to avoid coercions.

5.2 Comparison Operators

While occurrence typing is useful for predicates, programmers are more likely to employ comparison operators than predicates in numerical programs. Returning to the `abs` function, we can rewrite its body to use a comparison and it still type checks:

```
(if (> x 0) x (- x))
```

From the programmer's perspective, the two versions of the function are identical. The `>` function is not a predicate, however. We can still use the expressiveness of the types to encode this information in the type of `>`, leading to (among other overloadings):

$$\texttt{>} : \left(x : \texttt{Real}\ y : \texttt{Zero} \xrightarrow{\texttt{Positive}_x} \texttt{Boolean}\right)$$

That is, when the second argument (y) of `>` is `Zero`, the result of the comparison is true only if the first argument (x) is `Positive`.

Comparison with distinguished integer literals is a special case that appears only in a few types of the base environment. More commonly, comparison operators are used as in the `sum-vector` function given in figure 1. Its definition is

```
(define (sum-vector v)
 (define n (vector-length v))
 (let loop ([i 0] [sum 0])
   (if (< i n) (loop (+ i 1) (+ sum (vector-ref v i))) sum)))
```

There are several aspects of this definition to note. First, `n` is the result of `vector-length`, which must be of type `Index`. Second, `i` is initially `0` and it is only incremented, classifying it as a `Natural`. Now, when we consider the comparison operation, we see that if `(< i n)` is true, then `i` must be both greater than 0 and smaller than the largest possible vector, meaning that `i` must be an `Index` itself in the *then* branch. This is exactly the needed information to prove that `i` is *always* a `Fixnum`, allowing the compiler to optimize both the addition and comparison to use simple and efficient machine instructions.

To express this information, we again use the mechanism of associating propositions about argument types with the boolean result of functions:

$$\texttt{<} : \left(x : \texttt{Natural}\ y : \texttt{Index} \xrightarrow{\texttt{Index}_x} \texttt{Boolean}\right)$$

Further, this technique applies to comparison operators for all range-bounded types, such as `Fixnum`, as well as signed types such as `Negative`.

Using occurrence typing in conjunction with overloadings of comparison operators, Typed Racket can automatically prove tight bounds on numeric types based solely on the dynamic checks *already present* in programs. This supports both optimization and static checking for programs such as `sum-vector`.

6 Implementation

Over the past year, we implemented the type environment of sections 3 through 5 in Typed Racket without changing the basic type system. More precisely, the type assignment for the primitive operations now encodes basic mathematical theorems. Building a practical type system from these encodings has posed some challenges, however. We discuss the interesting ones in this section.

6.1 Precise Types and Invariance

While it is generally desirable to assign precise types that include sign and range information, doing so can sometimes lead to unexpected behavior. Consider the program

```
(define x (box 3))
(set-box! x 2000)
```

This program defines a mutable box that contains the integer `3`. The most precise type we can locally infer for `3` is `Positive-Byte`, and if we were willing to use this type, `x` could be assigned the type `(Boxof Positive-Byte)`. This type assignment implies, however, that attempting to set the contents of `x` to `2000` is a type error. Similar issues arise with any invariant type constructor.

Although this behavior is perfectly correct from a theoretical perspective, it has severe usability drawbacks. In the code bases we studied, initializing a box with a small integer, often zero, and later assigning significantly larger ones is a common occurrence. We therefore make this common case the default.

This decision means that the typechecker generalizes types that are used as arguments to invariant type constructors. In the above example, `Positive-Byte` would be generalized to `Natural`, and `x` would be of type `(Boxof Natural)`, which is more broadly useful. Generalization requires balancing of course. For example, we could use `Complex` instead of `Natural`, but doing so would discard all the information contained in the original type. The generalization function takes into account heuristics inspired by our corpus of numeric Racket programs as well as feedback from users of Typed Racket.

Finally, programmers can override the results of local type inference with explicit annotations to assign more permissive or restrictive types.

6.2 Precise Types and Arguments

Precise types make it possible to enforce interesting numerical properties, but it may be inconvenient to enforce them at all times. For example, we could restrict `vector-ref`, which indexes into a vector, to accept only indices of type `Index`, which are guaranteed to not exceed the length of the longest possible vector.

An experiment with this choice indicates, however, that it leads to severe usability issues in practice. Consider this variant of `sum-vector` from figure 1:

```
(define (sum-vector v)
  (define n (vector-length v))
  (let loop ([i (- n 1)] [sum 0])
    (if (> i -1) (loop (- i 1) (+ sum (vector-ref v i))) sum)))
```

This loop should produce identical results to the original version of `sum-vector`, despite iterating backwards over the input vector. In this case though, the index `i` cannot be assigned the `Index` type, since its value is -1 for the last iteration of the loop. If we enforce that `vector-ref` can only accept indices of type `Index`, this program would not type check and the programmer would have to rewrite the loop to appease the type checker. Our experience suggests that this typechecking failure is both confusing and frustrating to programmers.

To avoid such usability problems, our type system abides by Postel's law (Postel 1980) as a guiding principle for the types of the Typed Racket standard library. Library functions typically feature somewhat permissive argument types—`vector-ref` accepts `Integer` as an index, and errors if necessary—and the most precise return type possible. That way, the proof obligations do not overwhelm the programmer. And yet,

programmers can benefit from precise return types when they do want to enforce stricter properties in their own code.

Thus, if a program wants to communicate that vector indices can only be of type `Index`, it is possible:

```
(: picky-vector-ref : (∀ (X) (Vectorof X) Index → X))
(define picky-vector-ref vector-ref)
```

Since the new restrictive type is a subtype of the original type of `vector-ref`, the program typechecks just fine. This technique could also be used to statically enforce that the second argument of the division operator cannot be zero.

6.3 Printing Types

Encoding properties in types means types become large. Although manipulating, and operating on, such large types intuitively impacts type-checking time, we have not noticed a significant impact in practice. The large size of these types is problematic, however, when a programmer must see them.

Error reporting is the most important point of contact between programmers and the types of primitives. If a function is given arguments of the wrong type, an error message is displayed, along with the valid argument types of this function. By displaying the domains of the function, the error informs the programmer of what constitutes a valid argument to the function. As such, this type is useful information.

Unfortunately, the large number of cases in some numeric types causes an explosion in the size of error messages. An example of such an error message is shown in the left column of figure 3. Each of these domains are associated to a different return type: adding two `Byte`s results in an `Index`, adding two `Float`s results in a `Float`, and so on. It makes sense to have all these domains as part of the type, but this information does not belong in error messages. If a programmer passes a string to the `+` function, the error message should merely say that `+` accepts only numbers.

To reduce the extraneous information in error messages, the type checker filters out domains that are subtypes of other domains. In the above example, since all domains are subtypes of `Number`, only this last one is displayed, as shown in the right column of figure 3. The error message is just as informative and much easier to digest than the original. This heuristic also ensures that the type checker does not discard unrelated domains. For example, if a function has just two domains, `Integer` and `Float`, both are present in the error message because they are unrelated by subtyping. This is desirable because both branches carry useful information.

In addition, before filtering subsumed domains, we remove any domains that would lead to results that are inconsistent with the expected return type. For example, if the type checker expects an `Integer` as the result of an application of `+`, it can safely discard domains involving `Float` and `Complex`. After this initial filtering, it can remove subsumed domains as before. As a result, the error message that is shown when applying `+` with an expected type of `Integer` mentions only `Integer`.

The same techniques are used when printing types at the REPL, which is the other important point of contact between programmers and types. Full types can be displayed on demand if programmers want to explore them.

```
> (+ 1 "A")
Type Checker: No function domains
  matched in function application:
Domains:
  Zero Zero
  Zero Positive-Byte
  Byte Positive-Byte
  Byte Byte
        ... <snip 58 lines> ...
  Real Real
  Float-Complex Number
  Number Float-Complex
  Number Number
Arguments: Positive-Byte String
 in: (+ 1 "A")
```

```
> (+ 1 "A")
Type Checker: No function domains
  matched in function application:
Domains: Number Number
Arguments: Positive-Byte String
 in: (+ 1 "A")
```

Figure 3: Original typechecking error message versus simplified error message

6.4 Typechecking Literals

Finally, the fine-grained distinctions among types affect the type-checking of literals. When assigning a type to a literal, the typechecker needs to know where that literal falls with regards to the divisions discussed previously. Since the typechecker has access to the value of literals, this is for the most part straightforward. Portability between platforms complicates matters, however, and compiled Racket programs are portable. In particular, it is possible to typecheck and compile a program on one architecture and to run it on a different one. While the range of integers that fit within a byte is constant, the range of numbers that Racket stores as tagged machine integers is architecture-dependent. Hence, the typechecker must make conservative assumptions and assigns fixed-width integer types only if it is correct to do so on all architectures supported by Racket. For this reason, the `Index` type is limited to the closed interval $[0, 2^{28} - 1]$ and `Fixnum` is limited to $[-2^{30}, 2^{30} - 1]$.

6.5 Optimization

Numeric types guide compiler optimizations, and the Typed Racket compiler (Tobin-Hochstadt et al. 2011) is no exception. It reuses existing optimization technology with few major changes. Most of the time, the optimizer ignores the fine-grained distinctions, for example, the distinctions among the various subtypes of `Float`. Doing so gives the compiler a view similar to what optimizers would see in other typed languages, making the reuse of existing optimization techniques straightforward. Examples of numeric optimizations performed by the Typed Racket compiler are dispatch elimination, unboxing and arity-raising of complex number operations.[6]

[6] Wright and Cartwright (1997)'s typing efforts for Scheme-like languages, as well as those of others, lacked the expressive power to distinguish between different classes of numbers and to optimize numeric code in this fashion.

7 Related Work

Many dynamic languages, such as Common Lisp (Steele Jr. 1994), Scheme (Sperber et al. 2009) and Smalltalk (Goldberg and Robson 1983) provide numeric towers. They also allow for mixed-type arithmetic and dynamically moving from one level of the tower to another. As far as programmer convenience is concerned, they offer most of the benefits of Typed Racket. Due to their dynamic nature, however, these languages provide little in terms of static checking. Other dynamic languages such as Python, and Ruby provide mixed type arithmetic and a numeric tower, but with fewer types, typically omitting exact rationals, complex numbers, and sometimes arbitary size integers as well.

Languages in the SIMULA 67 (Dahl 1968) tradition such as Java (Gosling et al. 2005), C (ISO 1999) and C++ (Stroustrup 2000) provide static checking, but let programmers escape the type system. These languages provide mixed-type arithmetic for a small number of specific cases, but beyond that, programmers have to rely on labor-intensive operator overloading tricks in C++ or settle for an inconvenient notation. Typed functional languages, such as Haskell (Marlow 2010), Standard ML (Milner et al. 1997) and Ocaml (Leroy et al. 2010) provide static checking equivalent to numeric layers alone without subtyping. Haskell provides a large and extensible set of layers, but it does not support the sign and range properties of Typed Racket. Also, each of these languages have different stances on overloading. Ocaml does not provide any overloading for numeric operations. Programmers must choose between the `+` and `+.` operators depending on whether they are adding integers or floating-point numbers respectively. SML provides overloading in a small number of cases in the same way Java does. Finally, Haskell's type classes provide overloading, but disallow mixed-type arithmetic. Special handling of literals makes mixed-type arithmetic unnecessary in some cases, but in general explicit coercions between numeric types are necessary.

Finally, the Habit (Jones 2010) language is a dialect of Haskell for systems programming. Its type system enforces arithmetic properties about integers, provides a large variety of fixed-width integer types and aims to provide a large array of strong static guarantees. However, this significantly increases the proof obligation on the programmer. Although the kinds of guarantees Habit provides are valuable when writing highly reliable systems software, the costs of these guarantees are inconvenient for a general-purpose language. In addition, Habit focuses on integers and does not seem to provide support for interesting properties of other numeric layers.

8 Conclusion

To facilitate numeric programming in Typed Racket, we have supplemented an existing practical type system with a base type environment that supports rich specification, concise expression, static checking and effective optimization. The environment makes crucial use of several existing Typed Racket features: union types for defining a precise numeric hierarchy, function overloading via intersection types for expressing properties of operations, and occurrence typing for reasoning about predicates and comparisons.

Our design supports both the convenience of a numeric tower as found in untyped languages as well as the static checking available with modern typed languages. Additionally, we support strong specifications expressing sign, range, and layer information

about numeric values. Our Typed Racket implementation demonstrates the effectiveness of the approach both in typechecking existing code as well as providing static information for effective optimizations of numeric programs.

Bibliography

Peter Buneman and Benjamin Pierce. Union types for semistructured data. In *Proc. Works. on Database Programming Languages*, pp. 184–207, 1999.

Mario Coppo and Mariangiola Dezani-Ciancaglini. A new type assignment for λ-terms. *Archiv Math. Logik* 19, pp. 139–156, 1978.

Ole-Johan Dahl. *SIMULA 67 Common Base Language*. Norwegian Computing Center, 1968.

Matthew Flatt and PLT. Reference: Racket. PLT Inc., PLT-TR-2010-1, 2010. `http://racket-lang.org/tr1/`

Adele Goldberg and David Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, 1983.

James Gosling, Bill Joy, Guy L. Steele Jr., and Gilad Bracha. *The Java™ Language Specification*. Fourth edition. Addison-Wesley, 2005.

ISO. *ISO C Standard 1999*. 1999.

Mark P. Jones. The Habit programming language: the revised preliminary report. 2010.

Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system, Documentation and user's manual. 2010.

Peter A. W. Lewis, A. S. Goodman, and J. M. Miller. A pseudo-random number generator for the System/360. *IBM Systems Journal* 8(2), pp. 136–146, 1969.

Simon Marlow (editor). Haskell 2010 Language Report. 2010.

Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML, Revised Edition*. MIT Press, 1997.

Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems* 22(1), pp. 1–44, 2000.

Jon Postel. DoD standard Transmission control protocol. IETF RFC 761, 1980.

John C. Reynolds. Preliminary design of the programming language Forsythe. Technical report CMU-CS-88-159, Carnegie-Mellon University, 1988.

Michael Sperber, Matthew Flatt, Anton Van Straaten, R. Kent Dybvig, Robert Bruce Findler, and Jacob Matthews. Revised^6 report on the algorithmic language Scheme. *J. of Functional Programming* 19(S1), pp. 1–301, 2009.

Guy L. Steele Jr. *Common Lisp: the Language*. Second edition. Digital Press, 1994.

Bjarne Stroustrup. *The C++ Programming Language*. Third edition. Addison-Wesley, 2000.

Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *Proc. Symp. on Principles of Programming Languages*, pp. 395–406, 2008.

Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proc. International Conf. on Functional Programming*, pp. 117–128, 2010.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *Proc. Programming Language Design and Implementation*, pp. 132–141, 2011.

Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems* 19(1), pp. 87–152, 1997.