Experience Report: Applying Random Testing to a Base Type Environment

Vincent St-Amour
PLT @ Northeastern University
stamourv@racket-lang.org

Neil Toronto
PLT @ Brigham Young University
ntoronto@racket-lang.org

Abstract

As programmers, programming in typed languages increases our confidence in the correctness of our programs. As type system designers, soundness proofs increase our confidence in the correctness of our type systems. There is more to typed languages than their typing rules, however. To be usable, a typed language needs to provide a well-furnished standard library and to specify types for its exports.

As software artifacts, these base type environments can rival typecheckers in complexity. Our experience with the Typed Racket base environment—which accounts for 31% of the code in the Typed Racket implementation—teaches us that writing type environments can be just as error-prone as writing typecheckers.

We report on our experience over the past two years of using random testing to increase our confidence in the correctness of the Typed Racket base environment.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—Testing tools; D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Random testing, Type environments, Numeric towers

1. Types as Specifications

Typecheckers prove claims about the run-time execution of programs. Programmers and compilers depend on the validity of these claims when they reason about programs. The correctness of these claims depends not only on the correctness of the underlying type system, but also on the correctness of the base type environment of the language. The type for any given primitive operation must be consistent with the operation's run-time behavior.

In languages with sophisticated type systems, types can encode precise specifications for primitive operations. For example Alms (Tov and Pucella 2011) uses session types to encode mutual exclusion for lock operations, Vault (DeLine and Fähndrich 2001) uses typestate to encode protocols in the types of its networking primitives, and FISh (Jay 1999) uses shape types to encode the effect matrix operations have on matrix shapes. Such specifications can be subtle, making the encoding process error-prone.

ICFP'13, September 25–27, 2013, Boston, MA, USA.

ACM 978-1-4503-2326-0/13/09…$15.00.
http://dx.doi.org/10.1145/2500365.2500616

Typed Racket (Tobin-Hochstadt and Felleisen 2008) provides a rich numeric tower (St-Amour et al. 2012) similar to those provided by languages in the Lisp and Smalltalk families of languages, and it encodes precise type-based specifications for its numeric operations. The fine-grained types offered by the type system and the flexibility of the language's numeric operations result in large and complex specifications. Errors and oversights in these specifications have been a major source of bugs in Typed Racket.

The functional programming community has a successful history of using random testing (Claessen and Hughes 2000) to find bugs in programs ranging from undergraduate coursework (Page et al. 2008) to full-scale language models (Berghofer and Nipkow 2004; Klein et al. 2012; Klein et al. 2013). Inspired by these success stories, we decided to investigate whether random testing would help us detect bugs in the Typed Racket base environment—specifically in the implementation of its numeric tower. We report on our successful experience using PLT Redex (Felleisen et al. 2009) to build a random tester that we have been using for this task over the last two years.

The rest of the paper is organized as follows: section 2 provides background on the Typed Racket numeric tower, section 3 describes our random testing strategy, section 4 presents techniques we used to increase the effectiveness of our random tester, section 5 summarizes positive and negative aspects of our experiences and section 6 concludes.

2. The Typed Racket Numeric Tower

Typed Racket's approach to numeric types and operations—its *numeric tower* (St-Amour et al. 2012)—is inspired by those of untyped languages in the Lisp and Smalltalk families.

Numeric towers typically provide a deep hierarchy of numeric types including, in the case of Typed Racket, fixed-width and unbounded integer types, arbitrary-precision rational numbers, both single and double-precision floating-point numbers, and complex numbers. Numeric towers support generic and flexible numeric operations that accept numbers of any type, support *mixed-type arithmetic*—that is, programmers can freely mix and match numeric types such as `Integer` and `Float`—and always produce results with the most precise representation available.

In Typed Racket, the types of numeric operations include precise specifications of their flexible run-time behavior. Given the complexity of these behaviors, their specifications are often large and brittle.

The operations themselves are implemented in C as part of the Racket (Flatt and PLT 2010) runtime system and not in Typed Racket itself. This rules out the option of having the typechecker automatically infer—or even check—their types. The Typed Racket implementers must therefore manually assign types to each of them, a laborious and therefore error-prone process.

2.1 Specifications for Numeric Operations

In Typed Racket, numeric types can encode not only numeric *layers*—whether a number is an integer, a floating-point number, or a complex number—but also sign and range properties. For example, Typed Racket provides a `Positive-Float` type which only includes positive floating-point numbers and is a subtype of the more general `Float` type, which includes all floating-point numbers.

Types such as `Byte` and `Fixnum` impose range constraints on their members, the former including integers between 0 and 255 and the latter including all exact integers that can be represented in a tagged machine word. These types are related to each other via subtyping: `Byte` is a subtype of `Fixnum`, which is in turn a subtype of `Integer`—the type that contains all integers regardless of bit-width. Range-bounded types can be further partitioned by sign, to obtain for instance `Positive-Byte` or `Negative-Fixnum`.

Specifications for numeric operations—such as `+`, `round` and `sqrt`—can take advantage of the fine-grained distinctions encoded in these types to precisely describe the behavior of those operations. For example, the specification of the `sqrt` function expresses that the `Nonnegative-Real` type is closed under `sqrt`, but that it returns `Complex` numbers in the general case.

Types also encode the coercion rules that govern mixed-type arithmetic. For example, the specification of the `+` function expresses that the addition of an `Integer` and a `Float` returns a `Float`.

2.2 Encoding Specifications as Types

Typed Racket encodes specifications for numeric operations as ordered intersections of function types. That is, each property is individually encoded as a function type; all these properties are then intersected using Typed Racket's `case->` type constructor.

For example, the closure property of integers under addition is expressed using this function type,

```
[Integer Integer -> Integer]
```

the contagion rule for integer-float mixed-type arithmetic is expressed with these two function types

```
[Integer  Float    -> Float]
[Float    Integer  -> Float]
```

and sign preservation is encoded in these two function types

```
[Positive Positive -> Positive]
[Negative Negative -> Negative]
```

To build the type of `+` we intersect these properties, and all others that hold about addition:

```
(: + (case->
      [Zero     Zero     -> Zero]
      ...
      [Fixnum   Zero     -> Fixnum]
      [Zero     Fixnum   -> Fixnum]
      ...
      [Integer  Integer  -> Integer]
      ...
      [Integer  Float    -> Float]
      [Float    Integer  -> Float]
      [Float    Float    -> Float]
      ...
      [Positive Positive -> Positive]
      [Negative Negative -> Negative]
      ...))
```

Each side of a symmetric property needs to be encoded as a separate function type and, because of Typed Racket's checking rule for intersection types, additional function types are required for each case where two or more properties intersect. As a result, types for numeric primitives become large and complicated. For example, the type of `+` is an intersection type with 147 clauses while the type of `*` has 209 clauses.

3. Random Testing Strategy

Types for numeric operations have been a significant source of bugs in Typed Racket. Of the 576 reported Typed Racket bugs, 63 (10.9%) can be traced back to the types of numeric operations, and 39 others (6.8%) can be traced back to other parts of the base type environment. Such bugs signal a mismatch between the type of an operation and the run-time behavior of the operation. Since we consider the run-time behavior of the numeric primitives to be the correct semantics, types need to be fixed when bugs are found.

The Typed Racket developers have spent time and effort manually testing the type environment,[1] but manual testing alone is insufficient. Handwritten tests often either test properties in isolation, or test combinations of a small number of properties together. Most combinations remain unexplored, including some in which properties combine to create interesting corner cases. We used random testing to improve the coverage of our test suite and find bugs before they cause problems in the real world.

3.1 Random Testing with PLT Redex

PLT Redex (Felleisen et al. 2009) is an embedded domain-specific language—built inside of Racket (Flatt and PLT 2010)—for defining and mechanizing executable language models. A typical modeling workflow with Redex is as follows. First, users define a grammar and a reduction semantics for their language. Then, they write functions that encode properties that they expect to hold about terms in their language, such as type soundness (Tobin-Hochstadt and Felleisen 2008) or determinism (Kuper and Newton 2012). Afterwards, they write test cases to check whether the property holds for specific terms.

Redex also provides support for random testing (Klein and Findler 2010), inspired by QuickCheck (Claessen and Hughes 2000). Redex's random testing facility, `redex-check`, takes as input a language grammar and a property expressed about terms in that language, generates random terms and checks whether the desired property holds. It has been previously used to find mistakes in nine ICFP 2009 papers (Klein et al. 2012) and to find bugs in a model of the Racket virtual machine (Klein et al. 2013).

3.2 Testing Type Preservation

We designed a language of arithmetic terms using Redex. The structure of the language is simple: it consists only of arithmetic expressions, with numbers at the leaves and arithmetic operations as the internal nodes. To detect discrepancies between types and run-time behavior, we test type preservation on randomly generated terms in this language. The process is as follows:

1. Our tester generates a random arithmetic expression.
2. It then invokes the Typed Racket typechecker on the expression.
3. If the expression is ill-typed, the tester rejects it and starts again with a new expression.
4. Otherwise, the tester asks the Typed Racket implementation to evaluate the expression.
5. It invokes the Typed Racket typechecker to typecheck the result.

If the type of the result is not a subtype of the type of the original expression, then the claim made by the typechecker does not hold

[1] The Typed Racket test suite has over 10,000 lines of handwritten tests.

and the tester has found a discrepancy. The result is unsound, almost always due to a bug in the type environment.

We encoded this preservation property as a Racket function which `redex-check` then applies to random terms.

3.3 Example: Floating-Point Underflow Bug

To illustrate this process, let us examine an example bug we found using random testing. Consider this excerpt from the type we originally assigned to Racket's hyperbolic sine function, `sinh`:

```
(: sinh (case->
         [Float-Zero     -> Float-Zero]
         [Positive-Float -> Positive-Float]
         [Negative-Float -> Negative-Float]
         ...))
```

Mathematically, the hyperbolic sine function maps zero to zero, positive reals to positive reals, and negative reals to negative reals. The above type encodes this mathematical statement.

Now consider the following randomly generated term:

```
(sinh 1.2535e-17)
```

Since `1.2535e-17` is of type `Positive-Float`, the entire expression should be of type `Positive-Float`, given the type of `sinh` above. However, the term evaluates to `0.0` which, in Typed Racket, is of type `Float-Zero`. The type `Float-Zero` is not a subtype of `Positive-Float`, which includes only *positive* floating-point numbers. Thus, the type fragment above does not accurately reflect the runtime behavior of `sinh`.

For values of x sufficiently close to zero, $\sinh(x)$ is too small to represent as a floating-point number, causing an underflow. We therefore need to correct the above type to account for this subtle corner case. This revised type fragment, taken from the current type of `sinh`, addresses this issue:

```
(: sinh (case->
         [Float-Zero     -> Float-Zero]
         [Positive-Float -> Nonnegative-Float]
         [Negative-Float -> Nonpositive-Float]
         ...))
```

4. Improving Random Test Case Generation

Since we applied random testing to a specific domain—numeric operations—we were able to leverage domain constraints to improve the quality of randomly generated test cases. With higher-quality test cases, we were able to find more bugs with fewer attempts than by using Redex's test case generation directly.

We improved the quality of test cases in two ways. First, we increased the probability of generating well-typed terms. Second, we used specialized random number generation to generate numeric corner cases with higher probability.

4.1 Generating Well-Typed Terms

Testing type preservation makes sense only for terms that are well-typed in the first place; our test infrastructure automatically rejects ill-typed terms. Therefore, to minimize wasting time on generating unusable terms, we engineered our grammar to generate well-typed terms with high probability.

When first designing our term grammar, a key design decision was to only allow trees of literal numbers and arithmetic operations as expressions. Variables and binders do not appear in the grammar, and thus cannot appear in the generated terms. This not only guarantees that the generated terms are closed, but it also rules out—by construction—terms that contain variables whose types are inconsistent between definition and use sites. Since we are interested in testing the types of numeric operations, there is no loss of generality; adding binders and variables would not uncover bugs that would not have been found otherwise.

Strategy	% of rejected terms (lower is better)
Baseline	57.6%
+ integer non-terminal	45.3%
+ float non-terminal	41.1%
+ integer and float non-terminals	1.6%

Figure 1: Percentage of rejected terms by generation strategy

That choice proved effective in practice. As the first row of figure 1 shows, only around half of the generated terms were ill-typed using our first attempt at a grammar.

Looking at the ratio of rejected terms does not tell the whole story, however. Most numeric operations in Typed Racket accept any kind of number as input—the `+` function accepts, e.g., integers and complex numbers alike. Other operations—which we call *type-limited* operations—accept inputs only from a portion of the numeric tower. For example, `modulo` accepts only integers, and `fl+` is a floating-point specific addition operation. By looking at a sample of rejected terms, we noticed that terms containing type-limited operations accounted for most of the rejections.

Those rejected terms most often featured subexpressions of type `Float` (or one of its subtypes) used as arguments to integer-specific functions. Because the inexactness of floating-point numbers is "contagious" for most operations, randomly generated terms are more likely to evaluate to floating-point numbers than to integers. Less common were rejected terms containing float-specific operations receiving integers or complex numbers as arguments.

Since terms containing type-limited operations were more likely to be rejected than other terms, type-limited operations were under-tested compared to other operations. Types of type-limited operations are not, on average, significantly simpler than types of other numeric operations; testing type-limited operations adequately is thus important.

While the types of numeric operations distinguish input types at a fine-grained level—e.g. they distinguish by sign, range, etc.—they all possess a *most general domain* that is a supertype of all its other domains. For example, while the `+` function distinguishes between the addition of two positive integer and the addition of two negative integers and assigns different types to the results of the two expressions, it accepts any value of type `Number`, which is a supertype of all its other valid input types.

The most general domains of the vast majority of type-limited operations belong to a small set of types: `Integer`, `Float` and `Real`. Furthermore, the most general domains of most type-limited operations are closed under those operations. Armed with these two observations, we refactored our arithmetic expression grammar so that it included an additional non-terminal for each of these types.

The productions for these non-terminals include literal numbers of that type, as well as type-limited operations with the appropriate most general domain. In addition, these non-terminals include productions for other, non-type-limited operations on which the non-terminal's type is closed. For example, to test the portion of the type of `+` that handles values of type `Integer` and its subtypes, we added productions that generate addition terms to the integer non-terminal. Figure 2 shows an abbreviated version of the resulting grammar. The `define-language` form takes the name of the language as its first argument, and each subsequent clause defines a non-terminal, in which the first element of the clause is the non-terminal's name and the remaining elements are productions.

```
(define-language tr-arith

  ; Float and its subtypes
  [F* (random-float) F]
  [F (* F* F*) (+ F* F*) (- F* F*) (/ F* F*)
     (max F* F*) (min F* F*) (abs F*)
     (floor F*) (ceiling F*)
     (truncate F*) (round F*)
     (cos F*) (sin F*) (tan F*)
     ...
     ; float-specific operations
     (fl+ F* F*) (fl- F* F*)
     (fl* F* F*) (fl/ F* F*)
     (flmin F* F*) (flmax F* F*) (flabs F*)
     ...]

  ; Integer and its subtypes
  [I* (random-integer) I]
  [I (* I* I*) (+ I* I*) (- I* I*)
     (max I* I*) (min I* I*) (abs I*)
     ...
     ; integer-specific operations
     (modulo I* I*) (remainder I* I*)
     (quotient I* I*) (gcd I* I*) (lcm I* I*)
     ...]

  ; any numeric type
  [E* (random-number) E F I]
  [E (* E* E*) (+ E* E*) (- E* E*) (/ E* E*)
     (max E* E*) (min E* E*) (abs E*)
     (floor E*) (ceiling E*)
     (truncate E*) (round E*)
     (sqrt E*) (log E*) (exp E*)
     (cos E*) (sin E*) (tan E*)
     ...])
```

Figure 2: Abbreviated grammar for Typed Racket arithmetic expressions

This grammar refactoring significantly lowered the term rejection rate. Figure 1 shows the impact of the additional non-terminals on the rejection rate. In turn, this refactoring increased the amount of testing of type-limited operations.

4.2 Generating Numeric Corner Cases

In addition to reducing the ratio of rejected test cases, we were also interested in increasing the ratio of bugs found per test case. Most of the bugs we had found in Typed Racket's numeric types were related to *numeric corner cases*, that is, cases where the usual rules governing the behavior of numeric operations do not apply. Corner cases lead to counter-intuitive behavior, which we—as type environment designers—often overlook and forget to encode as part of numeric types.

Based on our corpus of numeric tower-related bugs, we identified three main sources of corner cases: mixed-type arithmetic, *special values* and *representation limitations*. A key observation is that corner cases are not uniformly distributed across the entire numeric tower. To increase the likelihood that individual test cases would trigger a bug, we engineered random number generation to generate these three cases on a regular basis, while still adequately testing the common cases.

4.2.1 Mixed-type arithmetic

Mixed-type arithmetic exercises the coercion and contagion rules of numeric primitives described in section 2.2. These rules are quite complex and not always intuitive, which makes the portions of numeric types that deal with mixed-type arithmetic especially error-prone.

To trigger mixed-type arithmetic, the random generator we use for real numbers picks a numeric layer at random, then calls the relevant layer-specific generator. Specifically, the `random-integer->random-real` function, shown in figure 3, takes as argument a random integer provided by the Redex random test generator and returns a random real number from a random layer.

Using this technique, random terms generated for the "any numeric type" non-terminal are highly likely to feature mixed-type arithmetic. Furthermore, the "any numeric type" expressions can have subexpressions drawn from type-limited expressions, which also introduces mixed-type arithmetic.

4.2.2 Special values

When it comes to special values, such as the floating-point infinities, NaN and the floating-point zeroes, numeric primitives deal with these as special cases, usually mirrored by special cases in numeric types.

Integer `0` is also often handled as a special case. For example, `0` is the null element of multiplication, and causes the `*` function to ignore all coercion and contagion rules. Instead it returns `0` directly, and that must be reflected in the type.

To ensure an adequate representation of special values, we allocate portions of the random distributions specifically to them. As the last three conditional clauses in figure 4 show, NaN and each of the two floating-point infinities are generated with probability 0.05 when generating floating-point numbers. The floating-point zeroes are similarly special-cased. Special values in other layers—such as integer `0`, and the single-precision floating-point special values—are handled similarly.

4.2.3 Representation Limitations

Because of limitations of numeric representations, properties that we expect to hold about mathematical functions may not hold of their implementations in the standard library. Since we rely heavily on our intuition about mathematical functions when assigning types and often overlook the specifics of numeric representations, such limitations are a frequent source of bugs.

Bugs due to overlooking floating-point underflow—as discussed in section 3.3—are examples of bugs due to representation limitations. Other examples include floating-point overflow and automatic fixnum to bignum promotion when integer values exceed a certain range.

To ensure that these cases are exercised, we reserve part of the distribution for values that are near representation boundaries—e.g. extremely small or extremely large floating-point numbers or integers just small enough to be represented as fixnums. Figure 4 also shows the generation of floating-point numbers in these ranges.

To quantify the effectiveness of these reserved portions of the distribution, we attempted to trigger the underflow bug discussed in section 3.3 by generating random applications of the `sinh` function. Without these reserved portions of our distribution, we generated five million random terms and none of them triggered the bug. With those reserved portions, we executed 10 test runs. For each run, the bug was always triggered within 29 attempts, with 7 attempts being neccessary on average.

```
(define (random-integer->random-real i)
  (define r (random))
  ; probability 0.25 each
  (cond
   [(< r 0.25)
    i] ; random integer
   [(< r 0.5)
    (random-integer->random-rational i)]
   [(< r 0.75)
    (random-integer->random-flonum i)]
   [else ; single-precision flonum
    (random-integer->random-single i)]))
```

Figure 3: Allocation of random numbers to layers

```
(define (random-integer->random-flonum i)
  (define r (random))
  (cond

    ; 0.25: laplace-distributed with scale i
    [(< r 0.25) (random-laplace (abs i))]

    ; 0.25: uniform bits
    [(< r 0.5)  (random-uniform-flonum)]

    ; 0.35: very small or very large
    [(< r 0.85)
     (define r (random))
     (cond [(< r 0.5)
            (define x (ordinal->flonum i))
            (if (= x 0.0)
                ; special case: float zeroes
                (if (< (random) 0.5) 0.0 -0.0)
                x)]
           [(< r 0.75)
            (flstep -inf.0 (abs i))]
           [else
            (flstep +inf.0 (- (abs i)))])]

    ; 0.05 each: +nan.0, +inf.0, -inf.0
    [(< r 0.9)   +nan.0]
    [(< r 0.95)  +inf.0]
    [else        -inf.0]))
```

Figure 4: Random generation of floating-point numbers

5. Lessons Learned

Randomly testing the Typed Racket base environment has been a success. We achieved our goal of automatically finding bugs in the Typed Racket environment. Based on our experience, we recommend using Redex-based random testing to typed language implementers looking to test their type environments.

This section provides some details on positive and negative aspects of our experience.

5.1 What Worked

Random testing shined for some aspects of our project. Of them, the following should be of general interest to other typed language implementers.

Finding bugs Our random tester successfully found a large number of bugs. These bugs had not been found through manual testing, which suggests that random testing and manual testing complement each other.

Some of the bugs uncovered by random testing had already been reported by users (but had not been fixed yet). Our random checker created test cases that were often more useful as starting points for the debugging phase than user-provided test cases. Randomly generated terms tended to be small compared to user-provided test cases, which were often complete functions. Furthermore, random test cases included numeric operations only which again narrowed down debugging significantly.

Testing the full implementation Our random tester calls out to the Typed Racket implementation directly both to typecheck terms and to evaluate them. Redex can call arbitrary Racket code, which made this workflow convenient. Being able to test the actual Typed Racket implementation, rather than testing a separate model, was key to effectively finding bugs.

Confidence when refactoring The Typed Racket numeric tower is in constant evolution. We need to assign new types when new primitives are added to the Racket runtime, and we constantly revise the design of Typed Racket's numeric tower to improve its usability. The addition of a random tester makes us much more confident when refactoring the numeric tower. We follow each change to numeric types with a round of random testing, in addition to manual testing.

For example, in July 2012, the first author undertook a major change in the design of the numeric tower: for usability reasons, we decided to include NaN as a valid value to all non-singleton floating-point types. This refactoring required changing the majority of the types of float-specific operations, and of many generic operations as well—several days of work. Without a random tester, the chances of introducing subtle bugs and not detecting them would have made such a refactoring terrifying. In the end, we may even have decided against it, and left the usability issue unaddressed.

With random testing as part of the development cycle, however, the refactoring was a success. We implemented a first version of the new types, ran the random checker, fixed the bugs it found, and repeated the process. When short test runs stopped revealing bugs, we let the checker run overnight. Eventually, we trusted our refactoring and released it. So far, no bug has been reported that could be traced to it.

Low effort required Writing the first version of the random tester took the first author, who had no previous experience with Redex, about two hours. That initial effort resulted in a working random generator that successfully found bugs. Racket's extensive `math` library made our subsequent experimentation with random distributions—described in section 4—easy.

Continuous integration Our random tester is run every time a change is committed to the Racket code base. DrDr,[2] Racket's continuous integration system, runs one thousand random test cases on every change, in addition to running the manually written Typed Racket test suite. Random testing accounts for about a minute and a half of this process.

In general, continuous integration is useful to detect regressions. When combined with random testing, it also increases test coverage by running new random test cases every time a change is made.

[2] `drdr.racket-lang.org`

5.2 What Could Have Worked Better

While we were developing our random tester, we encountered three limitations of Redex. None were show-stoppers—we successfully worked around them—but they nonetheless slowed us down.

Type system integration We needed to manually adjust our grammar to produce well-typed terms as often as possible. In doing so, we manually encoded aspects of the Typed Racket type system as part of our grammar. Changes to the type system may accidentally cause an increase of the test case rejection rate and require manual changes to the grammar. Had Redex been able to use our type system to guide its random generation, this effort would not have been necessary.

Default random number generation Redex provides a random number generator as part of its random testing support. It generates numbers from the full range of Racket's numeric types—including fractions and complex numbers—which made it a good starting point. Due to its inability to generate corner cases sufficiently often, however, we eventually replaced it with our own custom random number generator.

Test case reduction While randomly generated test cases were usually smaller than user-provided test cases, they were often larger than necessary. Often, only one or two of a test case's operations would be relevant to the exposed bug. Therefore, the first step of our response to failing test cases is to find which part of the term triggers the bug and isolate it. Had Redex provided test case reduction, our debugging cycle would have been significantly shorter.

6. Conclusion

Based on our experience with Typed Racket over the last two years, we have found that random testing is an effective technique for increasing confidence in a language's base type environment.

With only a small amount of effort, we were able to implement a basic random tester and uncover bugs. With a modest amount of additional work, we were able to increase the effectiveness of our tester significantly. As a result of those efforts, the Typed Racket base type environment is now much more robust than it was two years ago.

Acknowledgments We thank Casey Klein and Robby Findler for suggesting that we use random testing to find bugs in the Typed Racket numeric tower. This work has been supported by the NSERC.

Bibliography

Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *Proc. Conf. on Software Engineering and Formal Methods*, pp. 230–239, 2004.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proc. International Conf. on Functional Programming*, pp. 268–279, 2000.

Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. Conf. on Programming Language Design and Implementation*, pp. 59–69, 2001.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

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

C. Barry Jay. Programming in FISh. *International Journal on Software Tools for Technology Transfer* 2(3), pp. 307–315, 1999.

Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: on the effectiveness of lightweight mechanization. In *Proc. Symp. on Principles of Programming Languages*, pp. 285–296, 2012.

Casey Klein and Robert Bruce Findler. Randomized testing in PLT Redex. In *Proc. Works. Scheme and Functional Programming*, pp. 26–36, 2010.

Casey Klein, Matthew Flatt, and Robert Bruce Findler. The Racket virtual machine and randomized testing. *Higher-Order and Symbolic Computation*, 2013.

Lindsey Kuper and Ryan Newton. A lattice-theoretical approach to deterministic parallelism with shared state. Indiana University, TR702, 2012.

Rex Page, Carl Eastlund, and Matthias Felleisen. Functional programming and theorem proving for undergraduates: a progress report. In *Proc. Works. Functional and Declarative Programming in Education*, pp. 21–30, 2008.

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

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.

Jesse A. Tov and Riccardo Pucella. Practical affine types. In *Proc. Symp. on Principles of Programming Languages*, pp. 447–458, 2011.