Occurrence Typing Modulo Theories

Andrew M. Kent
Indiana University
andmkent@indiana.edu

David Kempe
Indiana University
dkempe@indiana.edu

Sam Tobin-Hochstadt
Indiana University
samth@indiana.edu

Abstract

We present a new type system combining *occurrence typing*, previously used to type check programs in dynamically-typed languages such as Racket, JavaScript, and Ruby, with *dependent refinement types*. We demonstrate that the addition of refinement types allows the integration of *arbitrary solver-backed reasoning* about logical propositions from external theories. By building on occurrence typing, we can add our enriched type system as an extension of Typed Racket, reusing the core of the existing formalism while increasing its expressiveness.

Dependent refinement types allow Typed Racket programmers to express rich type relationships, ranging from data structure invariants such as red-black tree balance to preconditions such as vector bounds. Refinements allow programmers to embed the propositions that occurrence typing in Typed Racket already reasons about into their types. Further, extending occurrence typing to refinements allows us to make the underlying formalism simpler and more powerful.

In addition to presenting the design of our system, we present a formal model of the system, show how to integrate it with theories over both linear arithmetic and bitvectors, and evaluate the system in the context of the full Typed Racket implementation[1]. Specifically, we take safe vector access as a case study, and examine all vector accesses in a 56,000 line corpus of Typed Racket programs. Our system is able to prove that 50% of these are safe with *no new annotation*, and with a few annotations and modifications, we can capture close to 80%.

1. From logical types to refinement types

Applying a static type discipline to an existing code base written in a dynamically-typed language such as JavaScript, Python, or Racket requires a type system tailored to the idioms of the language. When building gradually–typed systems, designers have accomplished this by aiming type systems at the simplest of type system goals—ruling out dynamic type errors such as trying to add together strings. These systems—ranging from widely-adopted industrial efforts such as TypeScript [15], Hack [8], and Flow [7] to academic systems such as Typed Racket [22], Reticulated Python [26], and Gradualtalk [2]—have been successful in this narrow goal.

```
(: max : [x : Int] [y : Int] ~>
   (Refine [z : Int]
      (and (>= z x) (>= z y))))
(define (max x y) (if (> x y) x y))
```

Figure 1: `max` with refinement types

However, advanced type systems have the power to express more powerful properties and to check more significant invariants than just the absence of dynamic type errors. Refinement types and dependent types, as well as sophisticated encodings in the type systems of languages such as Haskell and ML [10, 27], allow programmers to describe balanced binary trees, sized vectors, and much more, producing programs with stronger correctness guarantees.

In this paper, we combine these two strands of research, producing a system we dub *Refined Typed Racket*, or RTR. RTR follows in the tradition of Dependent ML [29] and Liquid Haskell [24] by supporting dependent and refinement types over a limited but extensible expression language. Experience with these languages has already demonstrated that a fully-decidable system can be quite expressive and describe rich program properties.

Furthermore, by building on the existing framework of *occurrence typing*, refinement types prove to be a natural addition to Typed Racket's implementation, formal model, and soundness results. Occurrence typing is designed to reason about type tests and control flow in existing Racket programs, using propositions about the types of variables and simple rules of logical inference. Extending this logic to encompass refinements of types as well as propositions drawn from solver-backed theories produces an expressive system which scales to real programs. In this paper, we show examples drawn from the theory of linear inequalities and the theory of bitvectors.

Figure 1 presents a simple demonstration of integrating refinement types with linear arithmetic. The `max` function takes two integers and returns the larger, but instead of giving it a simple type such as `(Int Int -> Int)`, as the

[1] Link to repository removed for double blind review.

current Typed Racket implementation specifies, we give a more precise type describing the behavior of `max`.

The syntax for refinement types in RTR changes `->` to `~>` and gives names to the parameters. We then use these names in the refinement of the result type, which must be at least as large as the inputs.

The function definition does not require any changes to accommodate the stronger type, nor do clients of `max` need to care that the type provides more guarantees than before. The refinement is simply a Racket expression, although the set of acceptable expressions is limited to those appropriate for the available theories (here linear arithmetic).

The conditional in the body of `max` enables the use of a refinement type in the result, as in most refinement type systems. However, Typed Racket's pre-existing ability to reason about conditionals means that abstraction and combination of conditional tests properly works in RTR without requiring anything more from solvers for various theories.

With these type system features we enable programmers to enforce new invariants in existing Typed Racket code and thus evaluate the effectiveness of our typechecker on real-world programs. As evidence, we have implemented our system in Typed Racket, including support for linear arithmetic and provably-safe vector access, and automatically analyzed three large libraries totalling more than 56,000 lines of code. We determined that approximately 50% of the vector accesses are provably safe with *no code changes*. We then examined one of the libraries in detail, finding that of the 75% not automatically proved safe, an additional 47% could be checked by adding type annotations and minor code modifications.

Our paper makes the following contributions:

1. We present the design of a novel and sound integration between occurrence typing and refinement types drawn from arbitrary logical theories.
2. We describe how to scale our design to a realistic implementation in the Typed Racket system.
3. We validate our design by showing how to verify the vast majority of vector accesses in a large Racket library with a small amount of effort using our implementation.

The remainder of this paper is structured as follows: section 2 reviews the basics of occurrence typing and introduces dependency and refinement types via examples; section 3 formally present the details of our type system, as well as soundness results for the calculus; section 4 describes the challenges of scaling the calculus to our implementation in Typed Racket; section 5 contains the results of our empirical evaluation of the effectiveness of using refinement types for vector bounds check verifications; section 6 discusses related work and section 7 concludes.

2. Beyond Occurrence Typing

Occurrence typing—an approach whereby different occurrences of the same identifier may be type checked at different types throughout a program—is the strategy Typed Racket uses to typecheck idiomatic Racket code [22, 23]. To illustrate, consider a Typed Racket function `least-significant-bit` which accepts either an integer *or* list of bits as input:

```
(: least-significant-bit :
   (U Int (Listof Bit)) -> Bit)
(define (least-significant-bit n)
  (if (int? n)
      (if (even? n) 0 1)
      (last n)))
```

Typechecking the function body begins with the assumption that `n` is of type `(U Int (Listof Bit))` (an ad-hoc union containing `Int` and `(Listof Bit)`). Typed Racket then verifies the test-expression `(int? n)` is well typed and checks the remaining branches of the program with the following insights:

- In the then branch we know `(int? n)` produced a non-`#f` value, thus `n` has type `Int`. The type system can then verify `(if (even? n) 0 1)` is well-typed at `Bit`.
- In the else branch we know `(int? n)` produced `#f`, implying `n` is not of type `Int`. This fact, combined with our previous knowledge of the type of `n`, tells us `n` must have type `(Listof Bit)` and thus `(last n)` is also well typed and of type `Bit`

This strategy of gleaning typed-based information from tests in control flow statements is an essential part of occurrence typing. Instead of describing the expression `(int? n)` as simply of type `Boolean`,

$$\vdash \texttt{(int? n)} : \mathbf{B}$$

our approach captures the type-based logical propositions implied by the truth of the result as well:

$$\vdash \texttt{(int? n)} : (\mathbf{B} \,;\, \mathbf{I}_n \mid \overline{\mathbf{I}_n})$$

These propositions state 'if the result is non-`#f` then $\mathbf{I}_n$ holds (meaning that `n` is an integer),' and 'if the result is `#f` then $\overline{\mathbf{I}_n}$ holds (`n` is not an integer).

These typed-based logical propositions and the usual logical connectives (conjunction, disjunction, etc ...) are already an integral part of Typed Racket's well-tested type system. By simplifying, enriching, and extending this foundation—adding extensible refinement types and new approaches for reasoning about aliasing and identifiers going out of scope—we can provide a more expressive, theory-extensible type system effective at verifying a wider array of practical compile-time program invariants.

2.1 Occurrence Typing with linear arithmetic and vectors

Consider how a standard vector access function `vec-ref` might be implemented in a relatively simply-typed language (e.g. Typed Racket today). In order to prevent the sorts of memory errors found in C-like languages, our function must conduct a runtime check before performing the raw, unsafe memory access at the user-specified index:

```
(: vec-ref :
   (∀ (α) ((Vecof α) Int -> α)))
(define (vec-ref v i)
  (if (⩽ 0 i (sub1 (len v)))
      (unsafe-vec-ref v i)
      (error "invalid vector index!")))
```

Although the straightforward type for `vec-ref` prevents some runtime errors statically, invalid indices remain a potential problem. In order to eliminate this source of errors, we can use our new system for occurrence typing with an extension for propositions from the theory of linear integer arithmetic (with a simple implementation of Fourier-Motzkin elimination [6] as a lightweight solver). This allows us to give $\leqslant$ a dependent function type where the truth-value of the result reports the intuitively implied linear inequality. We can then design a function `safe-vec-ref` to guarantee only provably valid indices are used:

```
(: safe-vec-ref :
  (∀ (α) ([v : (Vecof α)]
          [i : (Refine [i : Int]
                 (⩽ 0 i)
                 (< i (len v)))]
          ~> α)))
(define (safe-vec-ref v i)
  (unsafe-vec-ref v i))
```

While replacing *all* occurrences of `vec-ref` with `safe-vec-ref` in a program may seem desirable, with only that change, the likely result is code that no longer typechecks! This is because the validity of particular indices may not be apparent in scope of where they are used, or already present type-annotations in the program may override more specific types that are now available. For example, consider an attempt to use `safe-vec-ref` in a vector dot product function:

```
(: safe-dot-prod :
   (Vecof Int) (Vecof Int) -> Int)
(define (safe-dot-prod A B)
  (for/sum ([i (in-range (len A))])
    (* (safe-vec-ref A i)
       (safe-vec-ref B i))))
```

Typechecker error! Cannot prove (0 ⩽ i < (len B)) in expression: `(safe-vec-ref B i)`

Because there is no explicit knowledge about the length of `B`, `(safe-vec-ref B i)` will not typecheck as is: the type of `safe-dot-prod` must either be enriched to include the assumption that the vectors are of equal length, or a dynamic check must be added which verifies the assumption at runtime before entering the `for/sum` loop.

Safe vector access is a simple example of the program invariants expressible with occurrence typing extended with linear integer theory—we have chosen it for thorough examination, however, because it relates directly to our sizable case study on existing Typed Racket code. Xi [28], however, demonstrates at length how the invariants of far richer programs, such as balanced red-black trees and simple type-preserving evaluators, can be expressed using this same class of refinements.

2.2 Occurrence Typing with bitvectors

Another example of our new system leveraging an arbitrary solver-backed external theory can be seen in our usage of Z3's bitvector reasoning [16] for typechecking the helper function `xtime` found in many AES [1] encryption implementations. This function computes the result of multiplying the elements of the field $\mathbb{F}_{2^8}$ by x (i.e. polynomials of the form $\mathbb{F}_2[x]/(x^8+x^4+x^3+x+1)$ which AES conveniently represents using a byte):

```
(: xtime : Byte -> Byte)
(define (xtime num)
  (let ([n (AND (* #x02 num) #xff)])
    (cond
      [(= #x00 (AND num #x80)) n]
      [else (XOR n #x1b)])))
```

In this example the type `Byte` is actually shorthand for `(Refine [b : BitVector] (⩽ #x00 b #xff))` and we have enriched the type of the relevant bitwise operations to include propositions and refinements relating the values to bitvector-theoretic statements.

3. Formal Model

Our base system λ_{RTR} is a typed lambda calculus with a few primitives operations and base values, conditionals, and local bindings. An executable PLT Redex [9] model of this calculus is provided in the supplemental material.

The typing judgment for λ_{RTR} resembles the standard typing judgment except that instead of assigning just types, it assigns *type-results* to well-typed expressions:

$$\Delta \vdash e : [\tau \,;\, \psi_+ \mid \psi_-]$$

This judgment means that, in environment Δ:

- e has type τ.
- If e evaluates to a non-`false` (i.e. treated as true) value then the proposition ψ_+ (dubbed the "then proposition") holds.

$$
\begin{array}{rll}
p ::= & \texttt{not} \mid \texttt{add1} \mid \texttt{int?} \mid \texttt{zero?} \mid & \text{Primitive Ops} \\
& \texttt{bool?} \mid \texttt{proc?} \mid \texttt{pair?} \mid & \\
& \texttt{fst}^{\tau} \mid \texttt{snd}^{\tau} & \\
v ::= & n \in \mathbb{Z} \mid \texttt{true} \mid \texttt{false} \mid p & \text{Values} \\
e ::= & v \mid x \mid \lambda x^{\tau}.e \mid (\textbf{if}\ e\ e\ e) \mid (e\ e) \mid & \text{Expressions} \\
& (\textbf{let}\ (x\ e)\ e) \mid (\texttt{pair}\ e\ e) & \\
\varphi ::= & fst \mid snd & \text{Fields} \\
o ::= & x \mid (\varphi\ o) & \text{Objects} \\
\psi ::= & \tau_o \mid \overline{\psi} \mid \psi \wedge \psi \mid \psi \vee \psi \mid x \mapsto o \mid & \text{Propositions} \\
& \texttt{tt} \mid \texttt{ff} \mid \mathcal{X} \in \texttt{Th}_i \mid \exists x{:}\sigma.\ \psi & \\
\sigma, \tau ::= & \top \mid \mathbf{I} \mid \mathbf{T} \mid \mathbf{F} \mid \langle \tau \times \sigma \rangle \mid (\bigcup \overrightarrow{\tau}) \mid & \text{Types} \\
& x{:}\sigma \to \mathcal{R} \mid \{x : \tau \mid \psi\} \mid \exists x{:}\sigma.\ \tau & \\
\mathcal{R} ::= & (\tau\ ;\ \psi \mid \psi) & \text{Type-Results} \\
\Gamma ::= & \{\overrightarrow{x{:}\tau}\} & \text{Type Envs} \\
\Psi ::= & \{\overrightarrow{\psi}\} & \text{Prop Envs} \\
\Delta ::= & \Gamma; \Psi & \text{Environments}
\end{array}
$$

We use the following shorthands: $\bot$ for $(\bigcup)$, $\mathbf{B}$ for $(\bigcup\ \mathbf{T}\ \mathbf{F})$, and ν to range over metavariables of the form τ and $\overline{\tau}$.

Figure 2: Syntax of Types, Propositions, Terms, etc...

- If *e* evaluates to `false` then the proposition ψ_- (the "else proposition") holds.

3.1 Syntax

The syntax of types, propositions, terms, and other forms are given in figure 2.

Values in λ_{RTR} include integers, booleans, and primitive operations. Because λ_{RTR} omits polymorphism and type inference for brevity, we require explicit types on all usages of the pair accessors $\texttt{fst}^{\tau}$ and $\texttt{snd}^{\tau}$.

An **expression** may be a value, standard λ-abstraction, application, conditional, or let-expressions; we require as usual an annotation on an abstraction's argument. For simplicity we use an explicit `pair` expression for the construction of pairs.

Propositions in our system resemble those found in propositional logics with a few important domain specific additions. $\texttt{tt}$ and $\texttt{ff}$ are the trivial and contradictory propositions, $\overline{\psi}$ is standard logical negation, and $\wedge$ and $\vee$ represent the conjunction and disjunction of propositions. Type information is expressed by propositions of the form τ_o, which states that object o is of type τ. $x \mapsto o$ denotes the variable x is an alias for object o (i.e. they refer to the same value). An existentially quantified proposition of the form $\exists x{:}\sigma.\ \psi$ describes a proposition ψ which may depend on a variable x of type σ which is no longer in scope. Finally, an atomic propositions of the form $\mathcal{X}$ represents a statement from a theory $\texttt{Th}_i$ for which λ_{RTR} has been provided a sound decider. In this way our calculus describes an extensible system that can be enriched with various theories according to the needs of the application at hand.

Our **types** include a universal type $\top$ which describes all possible types. $\mathbf{I}$ is the type of integers, while $\mathbf{T}$ and $\mathbf{F}$ are the types of the values `true` and `false`. Pair types are written $\langle \tau \times \sigma \rangle$. $(\bigcup \overrightarrow{\tau})$ describes a 'true' (i.e. untagged) union of its components. Function types consist of a named argument x, a domain type τ, and range type-result $\mathcal{R}$ in which x is bound. $\{x : \tau \mid \psi\}$ is a standard refinement type, describing any value x of type τ for which ψ holds. Existentially quantified types of the form $\exists x{:}\sigma.\ \tau$ allow for types which may depend on an identifier no longer in scope, just as for propositions. Also, we assume refinements and pairs are kept in a normal form: nested refinements (i.e. a refinement of an already refined type) are collapsed (combining the refining propositions) and refinements in the fields pairs are instead lifted to be refinements on the pair which then refer to the respective field.

An **object** describes some accessible value in the current environment, which is either an in-scope identifier or a location within a data structure reachable via a series of **field** accessors (*e.g.* $(snd\ \ (fst\ \ x))$). Intuitively, objects are the pronouns of the statements in our logic.

Type-Results allow us to conveniently describe not only the traditional type of a term τ, but also what can be learned by testing its value in a conditional test-expression. If a term of type $(\tau\ ;\ \psi_+ \mid \psi_-)$ is found to be non-`false` then we can assume ψ_+ holds, otherwise the value is `false` and ψ_- holds. We often write $\exists x{:}\sigma.\ (\tau\ ;\ \psi_+ \mid \psi_-)$ to mean $(\exists x{:}\sigma.\ \tau\ ;\ \exists x{:}\sigma.\ \psi_+ \mid \exists x{:}\sigma.\ \psi_-)$ for convenience.

An **environment**, written Δ, is a pair consisting of a traditional type environment Γ and a proposition environment Ψ. As usual, Γ serves as a basic mapping from identifiers to types; Ψ is the set of currently known propositions. For convenience we write $\Delta, x{:}\tau$ to mean extending the Γ within Δ with the entry $x{:}\tau$ and Δ, ψ to mean extending the Ψ within Δ with the proposition ψ.

Syntactic elements of λ_{RTR} are well-formed with respect to an environment Δ only when each of their free variables has a corresponding entry in the type environment of Δ. All judgments and metafunctions in our system respect this well-formedness criteria.

3.2 Typing Rules

The typing judgment is defined in figure 3. We use the standard convention of choosing fresh names not currently bound in Γ when extending Γ with new bindings.

T-VAL is used for non-`false` values, assigning the type specified by δ_{τ} and then- and else-propositions consistent with the value being non-`false`. Conversely, T-FALSE assigns to `false` the corresponding type $\mathbf{F}$ and propositions declaring it impossible for the value to be non-`false`.

T-VAR consults a metafunction select-type$_{\Delta}(x)$ when typechecking a variable x in environment Δ, which may choose any appropriate type τ for x where τ_x is derivable in Δ. It then refines the type with an alias $\{y : \tau \mid y \mapsto x\}$, indicating the correspondence between this value and the

$$\text{T-VAL}\quad \frac{v \neq \texttt{false}}{\Delta \vdash v\ : (\delta_\tau(v)\ ;\ \mathtt{tt} \mid \mathtt{ff})}$$

$$\text{T-FALSE}\quad \Delta \vdash \texttt{false}\ : (\mathbf{F}\ ;\ \mathtt{ff} \mid \mathtt{tt})$$

$$\text{T-PAIR}\quad \frac{\Delta \vdash e_1\ : \tau_1 \qquad \Delta \vdash e_2\ : \tau_2}{\Delta \vdash (\texttt{pair}\ e_1\ e_2)\ : (\langle \tau_1 \times \tau_2 \rangle\ ;\ \mathtt{tt} \mid \mathtt{ff})}$$

$$\text{T-ABS}\quad \frac{\Delta, x{:}\tau \vdash e\ : \mathcal{R}}{\Delta \vdash \lambda x^{\tau}.e\ : (x{:}\tau \to \mathcal{R}\ ;\ \mathtt{tt} \mid \mathtt{ff})}$$

$$\text{T-APP}\quad \frac{\Delta \vdash e_1\ : x{:}\tau \to \mathcal{R} \qquad \Delta \vdash e_2\ : \sigma \qquad x \notin \text{fv}(\sigma) \qquad \Delta \vdash \sigma <: \tau}{\Delta \vdash (e_1\ e_2)\ : \exists x{:}\sigma.\ \mathcal{R}}$$

$$\text{T-IF}\quad \frac{\begin{array}{c}\Delta \vdash e_1\ : (\tau_1\ ;\ \psi_{1+} \mid \psi_{1-}) \\ \Delta, \psi_{1+} \vdash e_2\ : (\tau_2\ ;\ \psi_{2+} \mid \psi_{2-}) \\ \Delta, \psi_{1-} \vdash e_3\ : (\tau_3\ ;\ \psi_{3+} \mid \psi_{3-}) \\ \psi_+ = (\psi_{1+} \wedge \psi_{2+}) \vee (\psi_{1-} \wedge \psi_{3+}) \\ \psi_- = (\psi_{1+} \wedge \psi_{2-}) \vee (\psi_{1-} \wedge \psi_{3-})\end{array}}{\Delta \vdash (\textbf{if}\ e_1\ e_2\ e_3)\ : ((\bigcup \tau_2\ \tau_3)\ ;\ \psi_+ \mid \psi_-)}$$

$$\text{T-LET}\quad \frac{\begin{array}{c}\Delta \vdash e_1\ : (\tau_1\ ;\ \psi_{1+} \mid \psi_{1-}) \\ \psi_1 = (\overline{\mathbf{F}_x} \wedge \psi_{1+}) \vee (\mathbf{F}_x \wedge \psi_{1-}) \\ \Delta, x{:}\tau_1, \psi_1 \vdash e_2\ : \mathcal{R}\end{array}}{\Delta \vdash (\textbf{let}\ (x\ e_1)\ e_2)\ : \exists x{:}\{x : \tau_1 \mid \psi_1\}.\ \mathcal{R}}$$

$$\text{T-VAR}\quad \frac{\text{select-type}_\Delta(x) = \tau \qquad \Delta \vdash \tau_x \qquad \sigma = \{y : \tau \mid y \mapsto x\}}{\Delta \vdash x\ : (\sigma\ ;\ \overline{\mathbf{F}_x} \mid \mathbf{F}_x)}$$

We use $\Delta \vdash e\ : \tau$ when the propositions in the type-result are arbitrary.

Figure 3: Typing Rules

variable x. Then- and else-propositions are simple, stating the self evident fact that if x is found to equal `false` then x is of type $\mathbf{F}$, otherwise x is not of type $\mathbf{F}$.

T-ABS, the rule for checking lambda abstractions, checks the body of the lambda in the extended environment mapping x to τ. The type-result from checking the body is then used as the range for the function type, and the then- and else-propositions report the non-falseness of the value.

T-APP handles function application, first checking that e_1 and e_2 are well-typed individually. It ensures the type of e_1 corresponds to a function type [2] and that e_2's type is a subtype of the domain. The type-result for the application is the range of the function with an existential quantifier capturing the argument expression's *precise* type; this enables the function's range to depend on the type of its argument.

T-IF is used for conditionals, describing the important process by which information learned from test-expressions is projected into the respective branches. After ensuring e_1 is well-typed at any type, we make note of the then- and else-propositions ψ_{1+} and ψ_{1-}. We then extend the environment with the appropriate proposition, dependent upon which branch we are checking: ψ_{1+} is assumed while checking the then-branch and ψ_{1-} for the else-branch. The type of a conditional is simply the union of the types of the two branches. Similarly, the then- and else-propositions must capture the fact that, in the general case, it is impossible to know which branch the resulting value came from. A disjunction containing each possible path is used in both cases.

[2] For simplicity of presentation we ignore the possibility that e_1's type is a function type *within* a refinement.

T-LET first checks whether the expression e_1 being bound to x is well typed. When checking the body, the environment is extended with the type for x and a disjunction describing what may be learned if branching on the value of x. Since x will not be in scope outside the body, we must wrap the result of typechecking with an existential quantifier recording the known information for this local x.

T-PAIR checks the basic types for e_1 and e_2 and returns a pair-type with propositions reflecting the obvious non-`false` nature of the value.

3.3 Subtyping and Proof System

The subtyping and proof system use a combination of familiar rules from type theory and formal logic.

3.3.1 Subtyping

The subtyping relation, described in figure 4, is mostly straightforward.

All types are subtypes of themselves and of the top type, $\top$. A type is a subtype of a union if it is a subtype of any element of the union. Unions are only subtypes of a type if *every* member of the union is a subtype of that type. Function subtyping has the standard contra- and co-variance in the domain and range; in order to reason correctly about dependencies when checking the range, the environment is extended to assign x the more specific domain type. Pairs are subtypes when their respective fields are.

For refined types, we have three rules: S-REFWEAKEN states if τ is a subtype of σ in Δ then so is any refinement of τ; S-REFSUB and S-REFSUPER allow subtyping

$$\text{S-REFL}\quad \Delta \vdash \tau <: \tau \qquad \text{S-TOP}\quad \Delta \vdash \tau <: \top$$

$$\text{S-UNIONSUB}\quad \frac{\forall \tau \in \overrightarrow{\tau}.\ \Delta \vdash \tau <: \sigma}{\Delta \vdash (\bigcup \overrightarrow{\tau}) <: \sigma} \qquad \text{S-UNIONSUPER}\quad \frac{\exists \sigma \in \overrightarrow{\sigma}.\ \Delta \vdash \tau <: \sigma}{\Delta \vdash \tau <: (\bigcup \overrightarrow{\sigma})}$$

$$\text{S-REFWEAKEN}\quad \frac{\Delta \vdash \tau <: \sigma}{\Delta \vdash \{x : \tau \mid \psi\} <: \sigma} \qquad \text{S-REFSUB}\quad \frac{\Delta, x{:}\tau, \psi \vdash \sigma_x}{\Delta \vdash \{x : \tau \mid \psi\} <: \sigma}$$

$$\text{S-PAIR}\quad \frac{\Delta \vdash \tau_1 <: \tau_2 \qquad \Delta \vdash \sigma_1 <: \sigma_2}{\Delta \vdash \langle \tau_1 \times \sigma_1 \rangle <: \langle \tau_2 \times \sigma_2 \rangle} \qquad \text{S-EXISTS}\quad \frac{\Delta, x{:}\sigma \vdash \tau_1 <: \tau_2}{\Delta \vdash \exists x{:}\sigma.\ \tau_1 <: \tau_2}$$

$$\text{S-REFSUPER}\quad \frac{\Delta \vdash \tau <: \sigma \qquad \Delta, \tau_x \vdash \psi}{\Delta \vdash \tau <: \{x : \sigma \mid \psi\}} \qquad \text{S-FUN}\quad \frac{\Delta \vdash \sigma_2 <: \sigma_1 \qquad \Delta, x{:}\sigma_2 \vdash \mathcal{R}_1 <: \mathcal{R}_2}{\Delta \vdash x{:}\sigma_1 \to \mathcal{R}_1 <: x{:}\sigma_2 \to \mathcal{R}_2}$$

$$\text{SR-RESULT}\quad \frac{\Delta \vdash \tau_1 <: \tau_2 \qquad \Delta, \psi_{1+} \vdash \psi_{2+} \qquad \Delta, \psi_{1-} \vdash \psi_{2-}}{\Delta \vdash (\tau_1\ ;\ \psi_{1+} \mid \psi_{1-}) <: (\tau_2\ ;\ \psi_{2+} \mid \psi_{2-})}$$

Figure 4: Subtyping Rules

inqueries about refinements to be translated into their intuitively equivalent logical inquiries.

Since existentially quantified types are only used as a tool for typechecking, there is only an explicit subtyping rule: S-EXISTS. Intuitively, an existentially quantified type is a subtype of a type if the subtyping relation holds in the appropriately extended environment.

Finally, the subtyping relation for Type-Results relies on subtyping for the types and logical implication for the then- and else-propositions.

3.3.2 Proof System

Figure 5 describes the proof system for λ_{RTR}: a relatively simple, algorithmic propositional calculus with several domain specific rules for reasoning about types and external theories. Instances of substitution, written $[o/x]$, represent the standard capture-avoiding replacement of x with o.

Many rules—such as L-TRUE, L-ANDI, or L-ORE—are taken directly from propositional logic, providing basic introduction and elimination forms for the standard logical connectives. Similarly, the L-NOT rules allow for the standard simplifications of negated propositions.

Several type-specific rules allow us to utilize type-theoretic reasoning when deriving propositions. L-SUB allows us to prove o is of type τ when o's type in Γ is a subtype of τ. L-SUBNOT conversely lets us prove o is not of type τ when we know o is not some supertype of τ. L-BOT, like L-FALSE, allows us to draw any conclusion since $\bot$ is uninhabited.

L-UPDATE plays a key role in our system, allowing positive and negative type statements in Ψ to refine the types in Γ. Since an object o in a proposition may include an arbitrary number of field accesses into a variable, this rule separates the *path* of field accessors (path(o)) from the underlying identifier (var(o)), before proceeding to update the identifier's type in Γ. Roughly speaking, updating τ with σ computes $\tau \cap \sigma$ while updating τ with $\overline{\sigma}$ computes $\tau - \sigma$. A full description of update is found in figure 6.

Rules L-REFE, L-EXISTSPROPE, and L-EXISTSTYPEE provide the proof system with a natural way to 'unpack' nested information from within existentials and refinements; as expected, the existential instantiations must be done using variables which are fresh for the current environment.

The refinement introduction rule, L-REFI, allows the claim that an object is of a refined type to be derived in two steps: first we prove the object has the underlying type and then prove its refining proposition indeed holds.

L-ALIASE uses $x \mapsto o$ propositions to simplify the environment and goal. It replaces free occurrences of x with o ubiquitously and converts the type-entry for x in Γ into a proposition about o.

Finally, a theory-specific proposition ψ (i.e. a χ or logical combinations of χ's) can be discharged using L-THEORY. This rule queries the relevant solver, written $\Psi \vdash_{\mathtt{Th}_i} \psi$, with an appropriate solver-compatible transcription of the environment and goal. L-THEORYUNSAT is similar, stating if any particular theory can derive a contradiction from our Ψ then our current goal is provable.

3.4 Integrating Theories

Our system is designed so it can be enriched with an arbitrary external theory as long as a solver for it is provided. To do this, we first define the set of *theory-specific value terms* our theory discusses. In the case of bitvector theory, this is a simple inductive set with base {`bv`, `x`} (where `bv` represents a bitvector literal value) and inductive constructors matching our bitwise operations (`AND`, `XOR`, etc...).

We then define the set of *predicates* relevant to our theory, which are simple relations over the theory-specific value terms we have previously defined. Since the standard bitvector operations include a rich set of comparison operators, we can make due using the singleton set containing a predicate which describes bitvector equality {`bv=`}.

Once we have defined the propositions for our theory, we need only enrich any necessary types for literals and functions so propositions from our theory appear in our

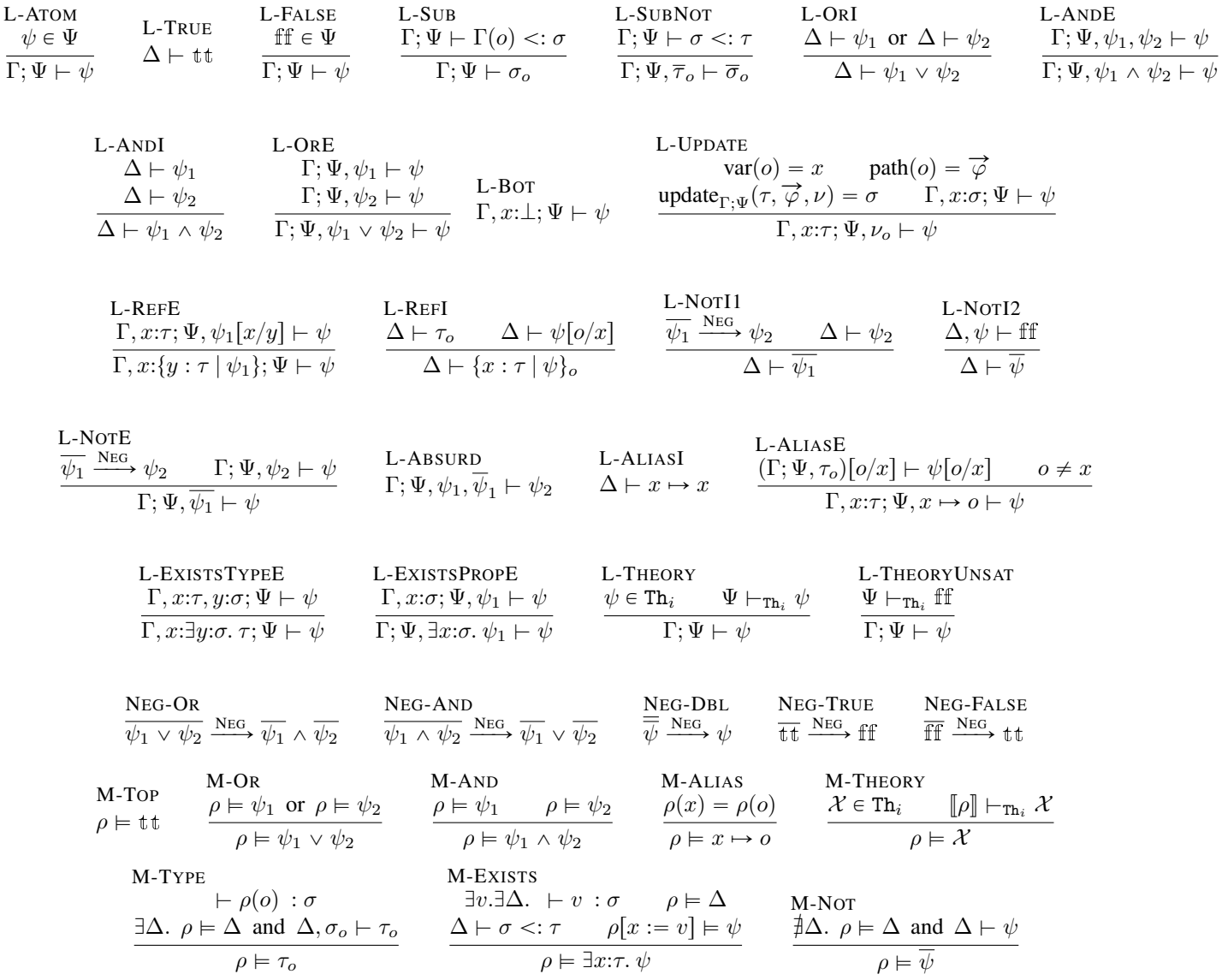

Figure 5: Logic Rules

environment while typechecking. For bitvectors, this includes extending δ_τ so a literal bitvector `bv` is assigned the *refined type* `(Refine [b : BitVector] (bv= bv b))` and so each bitwise operator has a type of the following form:

```
(: AND :
   [b1 : BitVector]
   [b2 : BitVector]
   ~> (Refine [b : BitVector]
              (bv= b (AND b1 b2))))
```

With these additions in place, a simple function converting bitvector propositions into Z3-compatible assertions allows our system to begin typechecking programs with bitvector-specific types.

3.5 Semantics and Soundness

λ_{RTR} uses an entirely standard big-step semantics which is included in the supplemental material. The evaluation judgment $\rho \vdash e \Downarrow v$ states that in runtime-environment ρ, expression e evaluates to the value v. A model-theoretic satisfaction relation, described in figure 5, is used to prove type soundness, just as in prior work on occurrence typing [23].

The metavariable ρ ranges over runtime-environments, which are finite maps from variables to closed runtime values. The $\rho(o)$ operation is a simple lookup for objects and accesses any fields in the obvious way.

We extend the set of values to include pairs and closures:

$$v ::= ... \mid \langle v\ ,\ v\rangle \mid [\rho, \lambda x^{\tau}.e]$$

3.5.1 Models

Because our system resembles a type-theory aware logic, it is convenient to examine its soundness using a model-theoretic approach commonly used in proof theory. For λ_{RTR} a model is any runtime-value environment ρ and is said to *satisfy* a proposition ψ, written $\rho \vDash \psi$, when its assignment of values to the free variables of ψ make the proposition inherently true. The details of satisfaction are defined in figure 5. The satisfaction relation extends to environments in pointwise manner, treating $x{:}\tau$ as the proposition τ_x. In order to complete our definition of satisfaction in figure 5, we also require a typing rule for closures and pair values, and subtyping for existentially quantified types (pair value typing is trivial and included in the supplemental material):

$$\text{T-CLOSURE}\ \frac{\exists\Delta.\ \rho \vDash \Delta \quad \Delta \vdash \lambda x^{\tau}.e\ :\mathcal{R}}{\vdash [\rho, \lambda x^{\tau}.e]\ :\mathcal{R}} \qquad \text{S-EXISTSR}\ \frac{\exists v.\ \vdash v\ :\sigma' \text{ and } \Delta \vdash \sigma' <: \sigma \quad \Delta, x{:}\sigma \vdash \tau_1 <: \tau_2}{\Delta \vdash \tau_1 <: \exists x{:}\sigma.\ \tau_2}$$

The satisfaction rules are mostly straightforward. $\mathtt{tt}$ is always satisfied, while the logical connectives $\vee$ and $\wedge$ are satisfied in the intuitive ways. A negated proposition is satisfied by ρ if it is impossible for a Δ satisfied by ρ to derive the proposition. Aliases are satisfied when the variable and object refer to the same value.

From M-TYPE we see propositions stating an object o is of type τ are satisfied when the value of o in ρ is a subtype of τ in some ρ-satisfied environment.

M-EXISTS appears complex but is quite natural once analyzed: an existentially quantified proposition is satisfied if we can extend ρ with an appropriate v—one which has the type τ in some ρ-satisfied environment—and show the underlying proposition holds in this extended environment.

Finally, the M-THEORY rule explains that ρ satisfies a proposition χ from theory $\mathtt{Th}_i$ when ρ's realization in $\mathtt{Th}_i$, written $[\![\rho]\!]$, implies χ according to the provided solver for $\mathtt{Th}_i$; the realization here is simply a transcription of the relevant variable assignments in ρ into sentences of the theory $\mathtt{Th}_i$.

3.5.2 Soundness

Our first lemma states that the proof theory for λ_{RTR} respects models.

Lemma 1. *If $\rho \vDash \Delta$ and $\Delta \vdash \psi$ then $\rho \vDash \psi$.*

Proof. By structural induction on $\Delta \vdash \psi$. □

With our proof theory and models in sync and our operational semantics defined, we can state and prove the next key lemma for type soundness which deals with evaluation.

$$\begin{array}{ll}
\text{update}_{\Delta}(\langle \tau \times \sigma\rangle, \overrightarrow{\varphi} :: fst, \nu) & = \langle \text{update}_{\Delta}(\tau, \overrightarrow{\varphi}, \nu) \times \sigma\rangle \\
\text{update}_{\Delta}(\langle \tau \times \sigma\rangle, \overrightarrow{\varphi} :: snd, \nu) & = \langle \tau \times \text{update}_{\Delta}(\sigma, \overrightarrow{\varphi}, \nu)\rangle \\
\text{update}_{\Delta}(\tau, \epsilon, \sigma) & = \text{restrict}_{\Delta}(\tau, \sigma) \\
\text{update}_{\Delta}(\tau, \epsilon, \overline{\sigma}) & = \text{remove}_{\Delta}(\tau, \sigma)
\end{array}$$

$$\begin{array}{lll}
\text{restrict}_{\Delta}(\tau, \sigma) & = \bot \text{ if } \nexists v.\ \Delta \vdash v : \tau \text{ and } \Delta \vdash v : \sigma & \\
\text{restrict}_{\Delta}((\bigcup \overrightarrow{\tau}), \sigma) & = (\bigcup \overrightarrow{\text{restrict}_{\Delta}(\tau, \sigma)}) & \\
\text{restrict}_{\Delta}(\tau, \sigma) & = \tau & \text{if } \Delta \vdash \tau <: \sigma \\
\text{restrict}_{\Delta}(\tau, \sigma) & = \sigma & \text{otherwise}
\end{array}$$

$$\begin{array}{lll}
\text{remove}_{\Delta}(\tau, \sigma) & = \bot & \text{if } \Delta \vdash \tau <: \sigma \\
\text{remove}_{\Delta}((\bigcup \overrightarrow{\tau}), \sigma) & = (\bigcup \overrightarrow{\text{remove}_{\Delta}(\tau, \sigma)}) & \\
\text{remove}_{\Delta}(\tau, \sigma) & = \tau & \text{otherwise}
\end{array}$$

Figure 6: Type Update

$$\begin{array}{ll}
\delta_{\tau}(n) & = \mathbf{I} \\
\delta_{\tau}(\mathtt{true}) & = \mathbf{T} \\
\delta_{\tau}(\mathtt{not}) & = x{:}\top \to (\mathbf{B}\ ;\ \mathbf{F}_x \mid \overline{\mathbf{F}_x}) \\
\delta_{\tau}(\mathtt{add1}) & = \mathbf{I} \to (\mathbf{I}\ ;\ \mathtt{tt} \mid \mathtt{ff}) \\
\delta_{\tau}(\mathtt{int?}) & = x{:}\top \to (\mathbf{B}\ ;\ \mathbf{I}_x \mid \overline{\mathbf{I}_x}) \\
\delta_{\tau}(\mathtt{zero?}) & = \mathbf{I} \to (\mathbf{B}\ ;\ \mathtt{tt} \mid \mathtt{tt}) \\
\delta_{\tau}(\mathtt{bool?}) & = x{:}\top \to (\mathbf{B}\ ;\ \mathbf{B}_x \mid \overline{\mathbf{B}_x}) \\
\delta_{\tau}(\mathtt{proc?}) & = x{:}\top \to (\mathbf{B}\ ;\ \bot \to \top_x \mid \overline{\bot \to \top_x}) \\
\delta_{\tau}(\mathtt{pair?}) & = x{:}\top \to (\mathbf{B}\ ;\ \langle \top \times \top\rangle_x \mid \overline{\langle \top \times \top\rangle_x}) \\
\delta_{\tau}(\mathtt{fst}^{\sigma}) & = x{:}\langle \sigma \times \top\rangle \to (\sigma\ ;\ \overline{\mathbf{F}_{(fst\ x)}} \mid \mathbf{F}_{(fst\ x)}) \\
\delta_{\tau}(\mathtt{snd}^{\sigma}) & = x{:}\langle \top \times \sigma\rangle \to (\sigma\ ;\ \overline{\mathbf{F}_{(snd\ x)}} \mid \mathbf{F}_{(snd\ x)})
\end{array}$$

Figure 7: Primitive Types & Operations

Lemma 2. *If $\Delta \vdash e\ : (\tau\ ;\ \psi_+ \mid \psi_-)$, $\rho \vDash \Delta$ and $\rho \vdash e \Downarrow v$ then all of the following hold:*

1. *$v \neq \mathtt{false}$ and $\rho \vDash \psi_+$, or $v = \mathtt{false}$ and $\rho \vDash \psi_-$*
2. *$\vdash v\ :\sigma$ and $\Delta \vdash \sigma <: \tau$*

Proof. By induction on the derivation of $\rho \vdash e \Downarrow v$. □

Now we can state our desired soundness theorem for λ_{RTR}.

Theorem 1. *(Type Soundness for λ_{RTR}). If $\vdash e\ :\tau$ and $\vdash e \Downarrow v$ then $\vdash v\ :\sigma$ and $\vdash \sigma <: \tau$.*

Although this model-theoretic proof technique works quite naturally, it includes the standard drawbacks of big-step soundness proofs, saying nothing of diverging or stuck terms.

4. Scaling to a real implementation

Although λ_{RTR} captures the essence the system, there are some important additional details to consider when reasoning about a realistic programming language.

4.1 Mutation

We use a basic approach for soundly supporting mutation in our type system. First, a preliminary pass identifies which variables and fields may be mutated during program execution. The typechecker then proceeds, recording the *initial* types for mutable variables in Γ but discarding any logical propositions learned about them in the body of the program.

To illustrate, consider a module variable `cache-size` of type `Int`. The type system will ensure any updates to `cache-size` are indeed sound. A function preparing to modify the cache may test `(> cache-size n)` to ensure it is of an appropriate size. It would be unsound, however, for our type system to assume the propositions learned from the test's result invariably hold; a concurrent thread could modify the cache and its size between our testing and performing the operation, invalidating our supposed guarantees.

This particular example was a latent bug discovered in the number-theory portion Racket's math library while performing our evaluation.

4.2 Intermediate Logical Simplifications

In order to highlight the quintessential features λ_{RTR} we neglected to require any intermediate logical simplifications during typechecking. This design decision would require potentially costly logical calculations to be repeated at numerous points while typechecking a program (e.g. when checking subtypes and selecting types for variables). If instead the typing rules eagerly performed even just the obvious simplifications when extending Ψ (e.g. eagerly updating Γ with the immediately available info), operations such as select-type$_{\Gamma;\Psi}(x)$ could be reduced to a constant type lookup $\Gamma(x)$ in many cases. This is promising when observing that in a larger implementation of this system, *expected types* [18] are often flowing down the syntax tree as well. In such cases, deciding whether $\Gamma(x)$ is sufficient or more work is required is reasonable. Our experience with Typed Racket suggests a middle ground between the λ_{RTR} approach and eager simplification is desirable.

Another valuable simplification we discovered that greatly reduced typechecking times for several Typed Racket programs was the decision to use a single representative member from the equivalence classes that arise from aliasing. By eagerly substituting and using a single representative member in the environment, large complex propositions which conservatively tracked dependencies—such as those arising from local-bindings—can be omitted entirely, resulting in major performance improvements for Typed Racket.

4.3 Extending type inference

Typed Racket uses an implementation of the standard local-type inference algorithm [18]. Since type inference is such an essential part of typechecking real programs, we were unable to check any interesting examples until we had accommodated refinement types.

The constraint generation algorithm in local type inference, written $\Gamma \vdash^{V}_{\bar{X}} S <: T \Rightarrow C$, takes as input an environment Γ, a set of type variables V, a set of unknowns $\bar{X}$ and two types S and T, and produces a constraint set C. Since the implementation of the algorithm already correctly handles when S is a subtype of T, we merely needed to add the natural cases which allow constraint generation to properly recurse into the types being refined:

$$\text{CG-Ref}\quad \frac{\Delta, x{:}\tau, \psi_1 \vdash \psi_2 \qquad \Delta \vdash^{V}_{\bar{X}} \tau <: \sigma \Rightarrow C}{\Delta \vdash^{V}_{\bar{X}} \{x : \tau \mid \psi_1\} <: \{x : \sigma \mid \psi_2\} \Rightarrow C}$$

$$\text{CG-RefLower}\quad \frac{\Delta \vdash^{V}_{\bar{X}} \tau <: \sigma \Rightarrow C}{\Delta \vdash^{V}_{\bar{X}} \{x : \tau \mid \psi\} <: \sigma \Rightarrow C} \qquad \text{CG-RefUpper}\quad \frac{\Delta, x{:}\tau \vdash \psi \qquad \Delta \vdash^{V}_{\bar{X}} \tau <: \sigma \Rightarrow C}{\Delta \vdash^{V}_{\bar{X}} \tau <: \{x : \sigma \mid \psi\} \Rightarrow C}$$

This naturally requires maintaining the full environment of propositions throughout the constraint generation process.

4.4 Complex macros

Racket's `for`-macros are a commonly used pattern for reducing boilerplate when iteration is desired. This example iterates `i` from `0` to `(sub1 (len v))`, summing each element of the vector:

```
(for/sum ([i (in-range (len v))])
  (safe-vec-ref v i))
```

Although initially appearing trivial to typecheck, its expansion, which is general to accommodate the plethora of features `for` supports, has a more problematic form:

```
(let ([start 0] [end (len v)] [init 0])
  (define (loop pos val)
    (if (< pos end)
        (let ([i pos])
          (loop (+ 1 pos)
                (+ val
                   (safe-vec-ref v i))))
        val))
  (loop start init))
```

With no annotations we are left to infer types for both the domain and range of the anonymous function. Initially, our local type inference chooses type `Int` for the argument `pos`. This might be perfectly acceptable in Typed Racket, since `Int` is a valid argument type for `vec-ref`. However, when verifying vector bounds, `Int` is too permissive: it does not express the loop-invariant that `pos` is always non-negative. To cope with this, we experimented with adding additional heuristic to our inference for anonymous lambda applications: if a variable is, directly or indirectly, used as a vector index within the function, we try the type `Nat` instead of `Int`. This type, combined with the upper-bounds check

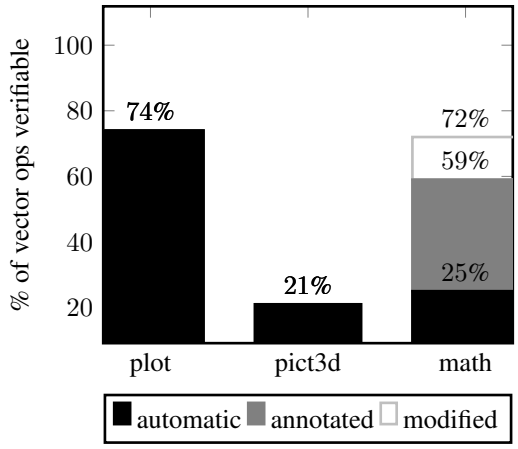

Figure 8: safe-vec-ref case study

within the loop, is enough for `(safe-vec-ref v i)` to typecheck.

In general, however, it is clear a better strategy for effectively inferring complex loop invariants required by some macros will be needed if patterns such as Racket's `for` are to be fully supported.

5. Case study: Safe vector access

In order to evaluate RTR's effectiveness on real programs we examined all vector accesses[3] in three large libraries written in Typed Racket, totalling more than 56,000 lines of code:

- The `math` library, a Racket standard library covering operations ranging from number theory to linear algebra. It contains 22,503 lines of code and 718 vector operations.
- The `plot` library, also a member of Racket's standard library, supporting both 2- and 3-dimensional plots. It contains 14,987 lines of code 1016 vector operations.
- The `pict3d` library,[4] which provides a purely functional interface to rendering hardware, is a performant, modern 3D engine. It contains 19,345 lines of code and 196 vector operations.

In essence, we tested whether each vector read and write could be replaced with its equivalent `safe-vec-` counterpart and still typecheck.

In total we verified approximately 50% of accesses without the aid of additional annotations. As figure 8 illustrates, our success rates for entirely automatic verification were 25% in `math`, 74% in `plot`, and 21% in `pict3d`. We attribute `plot`'s unusually high success rate (relative to the other libraries we tested) to a few heavily repeated patterns which are guaranteed to produce safe indexing: frequent pattern matching on vectors and loops using a vector's length as an explicit bound were extremely common.

For the `math` library we performed more thorough examination, classifying each individual access to determine how many of the failing cases our system might handle with reasonable effort:

- 34% of the failed accesses were verifiable after additional (or more specific) type annotations were added to the original program.
- 13% more were verifiable after small local modifications were made the to body of some functions. In some cases, these modifications added dynamic checks that allowed us to remove more checks elsewhere.
- 22% were unverifiable because, in their current form, their invariants were too complex to describe (i.e. they were outside the scope of our type system and linear integer theory).
- 6% involved Racket features we had neglected to support during implementation
- 2 calls made unsafe assumptions about a mutable cache whose size can shrink and cause errors at runtime.

In total, 72% of the vector accesses in the math library were verifiable with little or no changes to the existing programs and a majority of the changes involved annotations and slight modifications.[5]

For example, one function we encountered uses a loop to fill a vector with values extracted from a tree-like data structure. The loop is structured in such a way that the index is always assumed to be valid and the control flow is entirely dictated by the current shape of the tree. By restructuring this loop to use the same assumptions but instead explicitly test the current index to control the loop, we were able to safely remove all dynamic vector-related checks within the body of the loop without meaningfully changing the runtime-behavior (i.e. dynamic errors would be thrown in the same cases, but now by an initial check instead of by the first vector operation).

As we stated, however, some patterns were too complex for our system to verify, involving nonlinear arithmetic or dependencies not expressible in our calculus. An example of the latter is two lists of inherently related values: the first list containing vectors and the second containing sets of valid indices for the corresponding vector. This particular invariant might be successfully described if the relevant functions were refactored to reason about a single list with dependent pairs instead (i.e. the second element is a set of valid vector indices for the first element). However other problematic cases—such as those involving indices derived from non-trivial floating point calculations—seemed less amicable to verification with our available tools.

[3] Since Typed Racket reasons only about programs *after* macro expansion, vector accesses were assessed at this time as well.

[4] `https://github.com/ntoronto/pict3d`

[5] Link to repository removed for double blind review.

6. Related Work

There is a history of using refinements and dependent types to enrich already existing type systems. Dependent ML [29] adds a practical set of dependent types to standard ML to allow for richer specifications and compiler optimizations through simple refinements, using a small custom solver to prove constraints were satisfied. Liquid Haskell [24] extends Haskell's type system with a more general set of refinement types supported by an SMT solver and predicate abstraction. We similarly strive to provide an expressive, practical extension to an existing type system by adding dependent refinements. Our approach, however, seeks to enrich a type system designed *specifically* for dynamically typed languages and therefore is built on a different set of foundational features (*e.g.* subtyping, 'true' union types, type predicates, etc ...).

Some approaches, aimed at a higher bar of verification, have shown how enriching an ML-like typesystem with dependent types and access to theorem proving (automated and manual) provides both expressive and flexible programming tool. ATS, the successor of DML, supports both dependent and linear types as well as a form of interactive theorem proving for more complex obligations [3]. F* [20, 21] adds full dependent types and refinement types (along with other features) to an Fω-like core while allowing manual and SMT-backed discharging of proof obligations. Although our system shares the goal of allowing users to further enrich their typed programs beyond the expressiveness of the core system, we have chosen a simpler, less expressive approach aimed at allowing dynamically typed programs to gradually adopt a more practical set of dependent types.

Chugh et al. [5] explore how extensive use of refinement types and an SMT solver enable typechecking for rich dynamically typed languages such as JavaScript [4]. This approach feels most similar to ours in terms of features and expressiveness. As seen in our respective metatheories, however, their system requires a much more complicated design and a complex stratified soundness proof; a fact that has made it "[difficult to] add extra (basic) typing features to the language" [25]. In contrast, our system uses a well-understood core and does not *require* interaction with an external SMT solver. This allows us to use standard type-theoretic approaches and features—as witnessed by Typed Racket's continued adoption of new features.

Vekris et al. [25] explore how refinements can help reason about complex JavaScript programs utilizing a novel two phase approach which first establishes basic type properties while flagging ill-typed branches. A second pass then uses refinement types in the spirit of Knowles and Flanagan [14] and Rondon et al. [19] to verify ill-typed branches are in fact infeasible. Our system instead aims for a more unified and expressive approach, where an extensible set of refinements and their associated propositions naturally coexist with simple types and are both available in the surface syntax of the language.

Sage's use of a dynamic and static types is similar to our approach for typechecking programs. However, their usage of first-class types and arbitrary refinements means typechecking their core system is expressive yet undecidable [12]. Our system utilizes a more conservative, decidable core which allows for adding additional theories when further expressiveness is required. Also, our approach is not limited to a purely functional language.

Ou et al. [17] aim to make the process of working with dependent types more palatable by allowing fine-grained control over the trade-offs between dependent and simple types. This certainly is similar to our system in spirit, but there are several important differences. They choose to automatically insert coercions when dependent fragments and simple types interact, while we do not explicitly distinguish between the two and require explicit efforts in order to cast values. Additionally, while they convert their programs from a surface language into an entirely dependently typed language, our programs are translated into dynamically typed Racket code, which is void of any artifacts of our type system. This places us in a more suitable position for supporting sound interoperability between untyped and dependently typed programs.

Manifest contracts [11] are an approach that uses dependent contracts both as a method for ensuring runtime soundness and as a way to provide static typing information. Unlike our system, this method only reasons about explicit casts (i.e. program structure does not inform the type system) and there is no description of how a solver would be utilized to dispatch proof goals.

7. Conclusion

Enriching existing programs with stronger static guarantees is the original goal of the scripts-to-programs approach, realized in the type system underlying Typed Racket; this allows Racket programmers to add simple types to their programs with little effort. In this paper, we show how refinement types and an extensible logic allows programmers to continue this process by adding additional invariants to their repertoire which allow for strong new guarantees. Furthermore, our integration of refinement types with the occurrence typing underlying Typed Racket produces a new system more expressive and simpler than previous approaches.

Additionally, our evaluation demonstrates that despite the relatively simple nature of RTR's dependent types, the invariants that can be expressed are powerful. Our case study of vector operations finds that half of existing operations in a large Typed Racket code base can be already proved safe with most of the remainder checked with simple changes to the code. We anticipate that other programs, ranging from fixed-width arithmetic to theories of regular expressions [13], can similarly benefit from the strong specifications provided via refinement types. We look forward to continuing to enrich the specifications that Typed Racket programmers can describe.

Bibliography

[1] "Announcing the ADVANCED ENCRYPTION STANDARD (AES)." Federal Information Processing Standards Publication 197. United States National Institute of Standards and Technology (NIST). November 26, 2001.

[2] Esteban Allende, Oscar Callaú, Johan Fabry, Éric Tanter, and Marcus Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, 2013.

[3] Chiyan Chen and Hongwei Xi. Combining Programming with Theorem Proving. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 66–77, 2005.

[4] 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, 2012.

[5] 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, 2012.

[6] George B. Dantzig and B. Curtis Eaves. Fourier-Motzkin elimination and its dual. *J. Combinatorial Theory* 14(3), pp. 288–297, 1973.

[7] Facebook Inc. Flow: A static type checker for JavaScript. http://flowtype.org, 2014.

[8] Facebook Inc. Hack. http://hacklang.org, 2014.

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

[10] Matthew Fluet and Riccardo Pucella. Practical Datatype Specializations with Phantom Types and Recursion Schemes. In *Proc. Wksp. on ML*, pp. 203–228, 2006.

[11] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts Made Manifest. In *Proc. ACM Sym. Principles of Programming Languages*, pp. 353–364, 2010.

[12] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N. Freund, and Cormac Flanagan. Sage: Hybrid Checking for Flexible Specifications. In *Proc. Wksp. Scheme and Functional Programming*, pp. 93–104, 2006.

[13] David Herman and Philippe Meunier. Improving the static analysis of embedded languages via partial evaluation. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 16–27, 2004.

[14] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *Trans. Programming Languages and Systems* 32(2), pp. 6:1–6:34, 2010.

[15] Microsoft Co. TypeScript Language Specification. http://www.typescriptlang.org, 2014.

[16] Leonardo De Moura and Nikolaj Bjorner. Z3: An Efficient SMT Solver. In *Proc. TACAS*, 2008.

[17] Xinming Ou, Gang Ta, Yitzhak Mandelbaum, and David Walker. Dynamic typing with dependent types. *IFIP Conf. Theoretical Computer Science* 155, pp. 437–450, 2004.

[18] Benjamin C. Pierce and David N. Turner. Local Type Inference. *Trans. Programming Languages and Systems* 22(1), pp. 1–44, 2000.

[19] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid Types. In *Proc. ACM Conf. Programming Language Design and Implementation*, pp. 159–169, 2008.

[20] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure Distributed Programming with Value-Dependent Types. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 266–278, 2011.

[21] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-Monadic Effects in F*. In *Proc. ACM Sym. Principles of Programming Languages*, 2016.

[22] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage Migration: from Scripts to Programs. In *Proc. Dynamic Languages Symposium*, pp. 964–974, 2006.

[23] Sam Tobin-Hochstadt and Matthias Felleisen. Logical Types for Untyped Languages. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 117–128, 2010.

[24] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 269–282, 2014.

[25] Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. Trust, but Verify: Two-Phase Typing for Dynamic Languages. In *Proc. European Conf. Object-Oriented Programming*, pp. 52–75, 2015.

[26] Michael M. Vitousek, Andrew Kent, Jeremy G. Siek, and Jim Baker. Design and Evaluation of Gradual Typing for Python. In *Proc. Dynamic Languages Symposium*, pp. 45–56, 2014.

[27] Stephanie Weirich. Depending on Types. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 241–241, 2014.

[28] Hongwei Xi. Dependent ML: An Approach to Practical Programming with Dependent Types. *J. Functional Programming* 17(2), pp. 215–286, 2007.

[29] 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.