Practical Range Refinement Types with Inference

Valentin Aebi[0009−0000−7981−1743] and Carlo A. Furia[0000−0003−1040−3201]

Software Institute, USI – Università della Svizzera italiana, Lugano, Switzerland
{valentin.aebi,furiac}@usi.ch

Abstract. Refinement types are a static verification technique that aims at increasing the expressivity of traditional type systems while remaining easy and natural to use. While systems based on refinement types have been developed for several mainstream languages, their practical adoption remains limited by their annotation overhead, which is often a more significant burden than when using the "plain" type annotations of languages like Java or Scala.

To improve the state of the art, this paper introduces RANGER: a refinement type system designed to keep the annotation overhead small and to seamlessly integrate with imperative-style constructs like variables and loops. As the name suggests, RANGER focuses on integer *range types*: a particular kind of refinement types that express bounded integer ranges. Such types are widely useful to verify correct index manipulation and in-bounds data accesses, among others. To combine expressiveness and succinctness, RANGER is based on a *bidirectional* type system, which runs a type inference algorithm to provide the typechecking pass with information useful to reduce the need for user-written auxiliary annotations. RANGER also integrates other forms of lightweight flow-sensitive static analysis techniques that precisely capture the program's behavior without explicit annotations.

We implemented RANGER on top of the LICORNE experimental programming language. Our experiments show that RANGER's implementation can concisely express and verify a variety of useful properties that fall beyond the capabilities of standard static type systems like those of Java and Scala. The experiments also indicate that RANGER compares favorably to other extended type systems, such as the Java Checker Framework and Liquid Java, that can also check properties about ranges: RANGER can express more complex constraints, and usually requires no auxiliary annotations beyond those to specify a function's typed signature.

1 Introduction

Static type systems are an integral part of many programming languages, where they systematically rule out, at compile time, the presence of common programming errors. As such, they have earned a reputation as "the world's most successful formal method" [15]. Key to their success is a delicate balance between expressiveness and ease of use: the static type systems most commonly found in mainstream programming languages target a relatively narrow class of safety properties—such as operation compatibility and syntactic subtyping—that can be expressed and analyzed with only a reasonable amount of explicit user annotations. In contrast, statically verifying more complex, behavioral properties usually requires specialized tools and detailed annotations, which makes them impractical for casual or non-expert users.

Refinement types are an attempt to push the expressiveness of static type systems while retaining their ease of use. A refinement type uses a *predicate* to further constrain the set of values described by a type. In this paper, we focus on a category of refinement types that is widely useful: *integer range types*, which constrain integer values to be within certain bounds. For example, an array index variable should be between zero (included) and the array's length (excluded). As we discuss in Sec. 2, several frameworks have been developed in recent years to extend mainstream programming languages with support for refinement types. For example, the Checker Framework [25] and Liquid Java [14] offer annotations to refine Java types, both supporting at least some form of ranges. Our experiments described in Sec. 5, however, indicate that existing systems may not "play well" with some features of the language, require a significant number of additional user-written annotations, or have limited expressivity—so that expressing the "right" range type constraints is sometimes cumbersome or impossible.

In this paper, we introduce RANGER, a novel approach for range types with high precision and limited annotation overhead. The design of RANGER is based on two key principles: *i*) annotations should cover common programming idioms naturally; and *ii*) they should be succinct, requiring as little overhead as possible compared to "regular" type annotations. To this end, RANGER deploys complementary techniques: *i*) *bidirectional* typechecking [9] combines top-down type inference and bottom-up type-checking to reduce the amount of required annotations; *ii*) a *monotonicity* analysis improves the precision of type inference in the presence of loops; *iii*) *smart casts* enhance expressiveness by taking flow-sensitive[1] information into account; *iv*) a *hybrid cast* operator provides a runtime escape hatch in the few cases when the refinement types cannot be verified statically, which provides additional flexibility.

We implemented the RANGER technique as part of the type system of the LICORNE experimental programming language: a heavily redesigned variant of a prototype language we introduced in previous work [1]. Using LICORNE helped to quickly prototype our type system design and to integrate refinements as seamlessly as possible into the type system; however, the design of RANGER is general and independent of LICORNE, and could be implemented also in other programming frameworks with similar features.

Sec. 5 describes an experimental evaluation where we compared our implementation of RANGER to other refinement type frameworks that support similar range annotations, the Checker Framework and Liquid Java, as well as to the type system of Scala. These experiments indicate that RANGER is generally more succinct (requires fewer user annotations) and/or more expressive (can specify and check more complex constraints) than the other frameworks. These results suggest that RANGER achieves a practical trade-off between expressiveness and ease of use.

Contributions. This paper makes the following contributions:

- RANGER: a practical type system for range refinement types that combines expressiveness and succinctness thanks to advanced type inference;
- an implementation of RANGER, and an experimental comparison with other similar range refinement type frameworks;

[1] Typing rules for smart casts are actually *path sensitive*; however, since the typing literature usually simply designates such rules as "flow-sensitive" [5], we stick to this less precise terminology.

– for reproducibility, the prototype implementation of RANGER and all examples used in the experiments are available in a replication package: `https://doi.org/10.6084/m9.figshare.33022760`.

2 Related Work

Dependent types. Type systems are pervasive in programming and, more generally, computer science [26]. While most programmers are familiar with "vanilla" static type systems as they are used in mainstream programming languages, the full power of type systems goes well beyond distinguishing floating-point numbers from integers at compile-time. In fact, so-called *dependent* types—whose definition depends on program terms, such as a predicate in the case of refinement types—have been introduced as a way of formalizing logic reasoning that is amenable to automated reasoning [4,23], and underlie powerful interactive theorem provers such as Coq/Rocq [7] and Idris [3].

Refinement types. More recently, there has been a growing interest in applying refinement types to general-purpose programming languages while retaining ease of use. In particular, *liquid types* are refinement types whose predicates are restricted to a decidable logic fragment. Liquid types have been implemented on top of functional programming languages such as ML [12], Ocaml [29], and Haskell [33]—the latter arguably the most developed implementation of liquid types for a real-world programming language. Still in the realm of functional languages, Aeon [11] has supported refinement types from the very beginning of its development, focusing on their use for program synthesis.

Refinement types have also been developed for imperative languages. Flux [19] adds liquid types to Rust's ownership type system, allowing strong updates to change type predicates. Refined TypeScript [34] implements flow-sensitive refinement types for TypeScript by analyzing an SSA form—similar to RANGER's intermediate representation discussed in Sec. 4.3. Scala has also been extended with qualified types [30], a form of refinement types that is compatible with the language's functional and object-oriented features. Implementations of refinement types also exist for C [28] and Ruby [16].

The Java ecosystem also features frameworks that support refinement types, with an emphasis on language coverage and practical applicability. The Checker Framework [25] is a collection of extended typechecking plugins; one of them is the Index Checker [17] which aims at verifying array indexing (and other collection accesses) using range-like constraints. Recently, a form of liquid types has been implemented for Java [14], with a focus on usability. Our experimental evaluation in Sec. 5 will compare RANGER to similar frameworks in terms of expressiveness and degree of automation.

Type inference. RANGER's type system implements a form of bidirectional typing [9], a technique that combines top-down inference and bottom-up type-checking. As we detail in Sec. 4.3, RANGER's approach to bidirectional typing is however more global than the typical bidirectional typer, as we back-propagate types throughout every function.

3 Motivating Example

Fig. 1 shows the most significant parts of a program that demonstrates several of RANGER's capabilities. It is written in LICORNE (RANGER's current target language), whose syntax and semantics should be easy to glean for readers familiar with languages like Kotlin and Scala that combine functional and object-oriented features.

Function `decodeAll` in Fig. 1a implements the main functionality: given an integer array `data`, `decodeAll` converts each of its elements into a `Point`: a pair `x`, `y` of integer coordinates over an `xLen`-by-`yLen` bidimensional grid. To this end, `decodeAll` first filters the input `data` to discard all negative integers; then, it maps function `decode` onto each element. The actual conversion logic, implemented by `decode`, splits an integer into two parts: the most significant bits are interpreted as the `x` coordinate, whereas the 8 least significant bits become the `y` coordinate.

Range types. RANGER's range refinement type annotations are highlighted in green: *i*) Type `[1..]` requires `xLen` and `yLen` to be positive integers, so that both dimensions are valid; *ii*) The `x` and `y` components of the `Point` instances returned by `decodeAll` (in a list) and `decode` must be respectively of type `[0..<xLen]` and `[0..<yLen]`, that is integers between zero (included) and `xLen` or `yLen` (excluded); *iii*) Function `clamp` takes two integers `min` and `max` such that `max` ≥ `min` (i.e., `max` has type `[min..]`) and returns an integer in `[min..max]`; *iv*) The return type of `filter` applies the generic refinement operator `with` to parameter `T` to specify that the returned list only includes elements that satisfy filtering predicate `p`.

```
1  // Converts each integer in 'data' to a bidimensional point over a 'xLen'×'yLen' grid.
2  fn decodeAll(data: Array[Int], xLen: [1..], yLen: [1..]) → List[Point[[0..<xLen], [0..<yLen]]] {
3    val validData = filter(data.toList(), fn (d) → d >= 0);
4    return map(validData, fn (d) → decode(d, xLen, yLen));
5  }
6
7  // Converts 'datapoint' to a bidimensional point over a 'xLen'×'yLen' grid.
8  fn decode(datapoint: [0..], xLen: [1..], yLen: [1..]) → Point[[0..<xLen], [0..<yLen]] {
9    val x = clamp(0, xLen - 1, datapoint / 256);
10   val y = datapoint % yLen;
11   return new Point(x, y);
12 }
13
14 // Rounds 'x' to the closest integer in the range [min..max].
15 fn clamp(min: Int, max: [min..], x: Int) → [min..max] =
16   when x <= min then min else when x <= max then x else max
17
18 record Point[I sub Int, J sub Int](x: I, y: J)
```

(a) Function `decodeAll` converts an array of integers encoding coordinates into instances of record type `Point`.

```
19 fn filter[T](l: List[T], p: pure fn(T) → Bool) → List[T with p(it)] {
20   var rev = new Nil();
21   for (var rem = l; rem is Cons; rem = rem.rest())
22     { if p(rem.first()) { rev = new Cons(first=rem.first(), rest=rev); } };
23   return reverse(rev);
24 }
25
26 fn map[T, U](ls: List[T], f: fn (T) → U) → List[U] { /* ... */ }
27 fn reverse[T](ls: List[T]) → List[T] { /* ... */ }
28
29 class Array[T] {
30   pure fn length(this) → [0..] { /* ... */ }
31   fn at(this, i: [0..<this.length()]) → T  { /* ... */ }
32   fn toList(this) → List[T] {
33     var ls = new Nil();
34     for (var i = length() - 1; i >= 0; i -= 1) { ls = new Cons(first = at(i), rest = ls); };
35     return ls;
36   }
37 }
```

(b) Auxiliary functions and classes used by `decodeAll` to process lists and arrays.

Fig. 1: Motivating example.

Fig. 1's type annotations define fundamental correctness conditions of the program in a natural and succinct way. Even though range types are not sufficient to express full correctness, they still provide a rigorous documentation and rule out several potential errors. For example, typechecking the program verifies, among other things, that the `x` and `y` computed by `decode` fall in the expected ranges; and that `filter` returns a sublist of nonnegative integers, which matches `decode`'s requirement on its first argument `datapoint`.

Typechecking range types. Notably, RANGER can typecheck the program against its typed signatures without needing any annotations in the function *bodies*. To this end, RANGER deploys a bidirectional typechecking algorithm, which *infers* intermediate types in many cases. For example, to typecheck `filter`'s body, RANGER first back-propagates `filter`'s output type and records it as a type candidate for every assignment to `rev`; then, it performs a regular forward type analysis which validates the candidates and successfully checks the program. Typechecking `filter` also relies on *smart casts*, a feature that makes RANGER flow-sensitive: since the update of `rev` inside the body is guarded by condition `p(rem.first())`, RANGER can verify the conformance of the function to the predicate expressed by its return type: all elements of `rev` satisfy predicate `p`. This holds soundly only if `p` and `first` are pure functions. To this end, RANGER includes a purity enforcement mechanism, based on **`pure`** annotations such as the one in `p`'s typed signature: expressions used as type predicates or as path conditions in smart casts must be provably pure. Finally, typechecking the loop in method `toList` of class `Array` relies on another feature of RANGER: monotonicity analysis. The type-checker detects that `i` decreases across iterations, which, combined with smart cast based on the loop's staying condition $\texttt{i} \geq 0$, establishes that the array access is safe.

RANGER vs. other type systems. RANGER's refinement types can express properties that are out of reach for the type systems of mainstream languages with comparable features, including Java, Kotlin, and Scala. At the same time, as we detail in Sec. 5, the amount of user-written annotations required by RANGER is essentially the same as that of a regular Scala program. In other words, programmers can "upgrade" to RANGER without increasing their annotation burden. RANGER is not the only current implementation of refinement types for Java-like languages: as discussed in Sec. 2, the Checker Framework and Liquid Java provide comparable functionality for the Java language. The experiments detailed in Sec. 5 suggest that RANGER is often more flexible and expressive than the Checker Framework; for example, the latter cannot encode the constraint on `filter`'s return type in Fig. 1b. While LiquidJava's annotation language is quite expressive, it is limited in terms of language feature support; in our experiments, the LiquidJava implementation crashed on several examples and failed to find bugs in programs that use features like generics.

4 Design and Implementation of a Range Refinement Type System

The main *design goals* of the RANGER range-focused refinement type system are practicality and succinctness. This entails the following requirements: *i*) RANGER's type system should be *sound*; *ii*) it should be sufficiently expressive to cover *succinctly* the most *common* programming idioms that involve ranges; *iii*) in these scenarios, it should be *precise* without requiring user-written auxiliary annotations; *iv*) typechecking RANGER annotations should be efficient and deterministic.

4.1 Features of RANGER

Refinements and ranges. RANGER's type system extends a standard type system (similar to those available in languages like Scala and Kotlin) with a *refinement* type constructor T `with` p(`it`), which denotes a subtype of T consisting of all values $t \in T$ that also satisfy $p(t)$. Predicate p must be a pure boolean expression, that may depend on program values. While the refinement constructor is generic, RANGER's focus is on integer ranges. To this end, we introduce a shorthand for integer range types: `[l..u]` denotes the range from `l` to `u` (both included), and `[l..<u]` the range from `l` (included) to `u` (excluded). The notation also supports half-open intervals: e.g., `[0..]` denotes the nonnegative integers.[2]

Inference and analysis. RANGER should be usable with minimal auxiliary annotations. Ideally, users would only annotate function signatures, whereas the types of expressions within a function's body should be inferred by the type checker. To this end, RANGER implements a form of bidirectional typechecking: a backward type propagation runs *before* typechecking to infer type *candidates*, which are used in lieu of explicit programmer-written annotations. Type inference is best-effort and "aggressive": the typechecking phase is ultimately responsible for the soundness of typechecking, and will reject a type candidate if it cannot prove its validity or can infer a more precise type.

Precision and performance. To precisely check refinements, RANGER's typechecking algorithm also relies on external constraint solving. Since this dependency may significantly worsen performance, RANGER pre-processes the constraints sent to the external solver to make them tractable. As we better discuss in Sec. 4.4, solver constraints are quantifier-free (linear) arithmetic; in particular, analyzing loops does *not* introduce fixpoint computations.

Practicality. Among other features that make it more flexible, RANGER includes a hybrid cast operator `!!`, useful when statically checking a refinement is impractical. For example, consider an array access `a.at(k)`; if RANGER cannot determine that `k` is in-bound (e.g., because the access is guarded by a complex path condition), we can replace `k` with `k!!`. In this case, RANGER defers the check that `k` is in-bound to run time, while still statically checking `k`'s base type.

4.2 Notations and Preliminaries

IR syntax. We present RANGER's type system on programs written in IRCORNE: the intermediate representation that follows Fig. 2's syntax. This both makes the presentation more accessible and matches the actual implementation of RANGER, which also works on a (more complex) IR. IRCORNE is an SSA form, where every variable is assigned at most once and ϕ functions merge values from different control-flow paths. Unlike classic SSA representations, however, IRCORNE includes explicit structural control flow instructions (i.e., loops and conditionals), which are leveraged by RANGER's flow-sensitive analyses. Fig. 3 shows the IRCORNE translation of a simple function `maxPos` that computes the largest positive element in its input array `a`, which demonstrates IRCORNE's syntax in practice.

IRCORNE programs are collections of methods, whose body is a (possibly empty ϵ) sequence of instructions (i.e., a block, which also defines a scope). Instructions include assignments and ascriptions (checks that a variable has a given type), as well as control-flow instructions: conditionals, loops, returns, and `panic` (terminate abruptly

[2] Our range notation is inspired by the one used for range expressions in Kotlin and Groovy.

$$
\begin{array}{rll}
\text{Program } P & ::= & M^* \\
\text{Method } M & ::= & \texttt{fn}\ T_0.\texttt{fun}(v_1 : T_1, \ldots, v_n : T_n) \to T_{\text{ret}}\ \{\ S\ \} \\
\text{Block } S & ::= & \epsilon \mid D \mid A\ S \mid N\ S \\
\text{Control-flow } D & ::= & \texttt{if}\ c\ \texttt{then}\ S\ \texttt{else}\ S\ \texttt{merge}\ J^*\ \texttt{next}\ S & c \in \mathit{Vars} \\
 & \mid & \texttt{from}\ J^*\ \texttt{while}\ S(c)\ \texttt{loop}\ S\ \texttt{next}\ S \mid \texttt{return}\ v \mid \texttt{panic}\ v & c, v \in \mathit{Vars} \\
\text{Phi } J & ::= & v_{\text{out}} := \phi(v_{\text{left}}, v_{\text{right}}) & v_{\text{out}}, v_{\text{left}}, v_{\text{right}} \in \mathit{Vars} \\
\text{Assignment } A & ::= & v := e & v \in \mathit{Vars} \\
\text{Type ascription } N & ::= & v :: T & v \in \mathit{Vars} \\
\text{Type } T & ::= & B \mid B\ \texttt{with}\ q \mid T \sqcap T \mid T \sqcup T & \\
\text{Base type } B & ::= & \texttt{Any} \mid \texttt{Nothing} \mid \texttt{Unit} \mid \texttt{Int} \mid \texttt{Bool} \mid \texttt{String} \mid C & C \in \mathit{Classes} \\
\text{Predicate } q & ::= & p \mid l & p \in \mathit{Preds} \\
\text{Expression } e & ::= & v \mid v\ \texttt{as}\ B \mid v\texttt{!!} \mid c \mid l \mid a \mid \texttt{unit} & v \in \mathit{Vars} \\
\text{Call expression } c & ::= & \texttt{call}\ v_0.f(v_1, \ldots, v_n) & v_k \in \mathit{Vars} \\
\text{Logic expression } l & ::= & \texttt{true} \mid \texttt{false} \mid v_1 \diamond v_2 \mid v_1 \bowtie v_2 \mid !v_0 \mid v_0\ \texttt{is}\ B & v_k \in \mathit{Vars} \quad \diamond \in \{\wedge, \vee\} \quad \bowtie \in \{<, \leq, =\} \\
\text{Arithmetic expr. } a & ::= & n \mid v_1 \boxdot v_2 & n \in \mathbb{Z} \quad v_k \in \mathit{Vars} \quad \boxdot \in \{+, -, \times, /, \%\}
\end{array}
$$

Fig. 2: The syntax of IRcorne, licorne's intermediate representation. *Vars* stands for the set of program variables, each corresponding to a uniquely defined value in SSA form; *Preds* for the set of predicate identifiers (pure functions that return `Bool`), *Classes* for the set of user-defined types, and $\mathbb{Z}$ for the set of integer numbers. `Nothing` and `Any` are the bottom and top types.

```
fn maxPos(a: Array[Int]) → [0 .. ] {
  var k = 0; var max = 0;
  while (k < a.length()) {
    var v = a.at(k);
    if (v > 0 && v > max) {
      max = v;
    };
    k += 1;
  };
  return max;
}
```

(a) Function `maxPos` in licorne.

```
fn maxPos(a: Array[Int]) → [0..] {
  k0 := 0 ; max0 := 0
  from k1:=φ(k0,k2) ; max1:=φ(max0,max3)
  while tmp0 := call a.length() ; c0 := k1 < tmp0 (c0) loop
    v0 := call a.at(k1)
    tmp1 := v0 > 0 ; tmp2 := v0 > max1 ; c1 := tmp1 ∧ tmp2
    if c1 then max2 := v0 else ϵ
    merge max3 := φ(max2,max1)
    next k2 := k1 + 1
  next return max1
}
```

(b) Function `maxPos` in the IRcorne SSA intermediate representation.

Fig. 3: Function `maxPos` returns the largest non-negative element in array `a`, or zero if all elements of `a` are negative.

with an error). A conditional `if` c `then` t `else` e `merge` m `next` n introduces two blocks t and e, one of which executes depending on the Boolean condition c; in addition, a sequence m of ϕ functions merge the variables defined in the two branches; and n declares the block executed afterwards. A loop `from` m `while` $w(c)$ `loop` b `next` n executes a block b as long as a condition c is true; correspondingly, m is a sequence of ϕ functions that merge the variables in any predecessor block (including the back-edge from the loop's previous iteration); w are instructions that evaluate the condition c; and n determines the block executed after the loop exits. The `next` blocks make IRcorne a *functional* SSA form [2], where the targets of branching instructions are explicitly available.

Notations. We use the symbols $\sqcup$ and $\sqcap$ for union and intersection types; $T_1 <: T_2$ denotes that T_1 is a subtype of T_2. We also use $\langle A_k \rangle^k$ to denote the sequence $A_1\ A_2\ \ldots$ of A_k repeated for every value of k.

4.3 How ranger Works

Flattening. A disadvantage of the SSA form is that it breaks down complex expressions into assignment sequences, which complicates analyses that need whole source

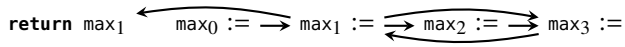
RULE	INSTRUCTION i	$\text{CND}_{i,\text{in}}$	$\text{CND}_{i,\text{out}}(x)$
exit	`return` v	$\text{CND}_{i,\text{out}}[v \mapsto \{T_{\text{ret}}\}]$	$\emptyset$
call	r `:= call` $v_0.m(v_1, \ldots)$	$\text{CND}_{i,\text{out}}\left\langle [v_k \mapsto \text{CND}_{i,\text{out}}(v_k) \cup \{T_k\}] \right\rangle^{k}$	
assign	$x := y$	$\text{CND}_{i,\text{out}}[y \mapsto \text{CND}_{i,\text{out}}(y) \cup \text{CND}_{i,\text{out}}(x)]$	$\bigcup_{j \in \text{succ}(i)} \text{CND}_{j,\text{in}}(x)$
join	$v := \phi(x_1, x_2)$	$\text{CND}_{i,\text{out}}\left\langle [x_k \mapsto \text{CND}_{i,\text{out}}(x_k) \cup \text{CND}_{i,\text{out}}(v)] \right\rangle^{k=1,2}$	
skip	any other instruction	$\text{CND}_{i,\text{out}}$	

Table 1: Candidate types inference rules. Each rule defines the set $\text{CND}_{i,\text{in}}$ of candidate inferred types before instruction i in terms of the set $\text{CND}_{i,\text{in}}$ that hold after instruction i.

expressions. To address this issue, we record the origin of each expression and make this information available to RANGER's analysis, which can use it to recover the *flat* form of every variable v, i.e., the complete expression whose value is stored by v. For instance, the flat form of $\texttt{c}_1$ in Fig. 3b is $\texttt{v}_0 > \texttt{0} \ \wedge \ \texttt{v}_0 > \texttt{max}_1$.

Candidate types inference. RANGER performs a backwards analysis that propagates types known from ascriptions, declared method return types, and call arguments, to help the subsequent typechecking phase with type synthesis. Formally, candidate types inference is a modular backward dataflow analysis, run individually on every method of the program. For each instruction i, the analysis computes a mapping $\text{CND}_i : \mathit{Vars} \to 2^T$ such that $\text{CND}_i(v)$ is the set of inferred types of variable v before i. Concretely, the dataflow analysis tracks the inferred types after ($\text{CND}_{i,\text{out}}$) and before ($\text{CND}_{i,\text{in}} = \text{CND}_i$) every i. The rules in Tab. 1 define how $\text{CND}_{i,\text{in}}$ is updated working backwards from $\text{CND}_{i,\text{out}}$: *i*) If i is an exit point (rule exit), $\text{CND}_{i,\text{out}}(x)$ is the empty set, which carries no information about x's type; then, $\text{CND}_{i,\text{in}}$ uses the declared return type T_{ret} as the candidate type of the returned variable. *ii*) If i is a call (rule call), the analysis tries to resolve the called method m, and then uses its declared parameter types T_k as new candidates for the types of the actual argument variables; if m cannot be resolved, no candidates are added. *iii*) If i is an assignment of y to x (rule assign), $\text{CND}_{i,\text{in}}(y)$ is updated based on the inferred type $\text{CND}_{i,\text{out}}(x)$ of x. *iv*) If i joins two variables with a ϕ function (rule join), the inferred type $\text{CND}_{i,\text{out}}(v)$ of the unified variable is propagated to both source variables. *v*) In all other cases, $\text{CND}_{i,\text{out}}(x)$ is the union of all candidate types inferred in the *successor* nodes j of i; and $\text{CND}_{i,\text{in}} = \text{CND}_{i,\text{out}}$, as the inferred candidates are simply backward propagated.

In Fig. 3's example, RANGER's type inference algorithm assigns type `[0..]` to every variant $\texttt{max}_k$ of variable `max`. Consider the control flow of the instructions involving `max`:



The analysis starts at the `return` exit point, where it infers type `[0..]` for $\texttt{max}_1$; from there, it goes backwards into the loop's head through the ϕ assignment to $\texttt{max}_1$, which infers the same type for $\texttt{max}_0$ and $\texttt{max}_3$; and then traverses the loop body until it reaches the ϕ assignment to $\texttt{max}_3$, where it infers `[0..]` also for $\texttt{max}_2$.

Monotonicity detection. RANGER analyzes loops to detect integer variables that are updated *monotonically* across loop iterations. The output of this analysis is a partial mapping $M : \mathit{Vars} \to \mathit{Ranges}$ of variables to range types. As discussed later, RANGER's type analysis uses this information to refine the inferred type of loop variables. Formally, monotonicity detection processes every ϕ assignment $v_{\text{next}} := \phi(v_{\text{init}}, v_{\text{prev}})$ in a loop's `from` clause, which merges the values of variable v set before the loop (v_{init}) and during the previous iteration (v_{prev}). RANGER constructs two monotonicity constraints: $\mu_\uparrow \triangleq f(v_{\text{prev}}) \geq v_{\text{next}}$ and $\mu_\downarrow \triangleq f(v_{\text{prev}}) \leq v_{\text{next}}$, where $f(v_{\text{prev}})$ denotes the *flat* form of

v_{prev} described above. RANGER uses an SMT solver to check the validity of the monotonicity constraints: if $\mu_\uparrow$ is valid, v_{next} is monotonically nondecreasing, and hence $M(v_{\text{next}})$ is the type `[` v_{init} `..]`; if $\mu_\downarrow$ is valid, v_{next} is monotonically nonincreasing, and hence $M(v_{\text{next}})$ is the type `[..` v_{init} `]`; if both are valid, v_{next} is constant, and hence $M(v_{\text{next}})$ is the type `[` v_{init} `..` v_{init} `]`. In all other cases (including if the flat form is unavailable), $M(v_{\text{next}})$ is undefined, so that monotonicity detection will always be sound.

In Fig. 3's example, RANGER determines that $\mathtt{k}_1$—set in the loop's head—is monotonically non-decreasing: the flat form of $\mathtt{k}_2$ is $\mathtt{k}_1 + 1$, and the constraint $\mu_\uparrow \triangleq \mathtt{k}_1 + 1 \geq \mathtt{k}_1$ is obviously valid; hence, $\mathtt{k}_1$ has type `[` $\mathtt{k}_0$ `..]`.

Type checking. Fig. 4 shows RANGER's typing rules, which we illustrate in the following. The rules consist of typing judgments $\vdash$ using an environment Γ that is a sequence of pairs $v : T$—denoting that variable v has type T.

Instructions. The type rules for instructions in Fig. 4a use typing judgments $\Gamma \vdash S \Uparrow T, \tau$, denoting that all `return` instructions in block S return a value of type T; τ is a Boolean that is $\top$ iff all paths through S end with a terminating instruction. For readability, we omit τ in rules that are not affected by it. Thus, rule function simply reduces checking a function to checking its body. Rules return, panic, and empty establish the typing judgments at the terminal points of each path. Rule ascription expresses an ascription as a type check. Rule assignment binds the type of source expression *e and* the associated control-flow information to the assignment target v, while forwarding return type T_{ret} and termination information τ.

Branching. The rules for loops and conditionals adapt [32]'s formalization of occurrence typing to model smart casts: c : `Bool`$;\psi^+|\psi^-$ means that ψ^+ holds if c is true, and ψ^- holds if c is false. Rule loop uses this information to check loops: *i*) the type T_k of variables $v_{k,\text{init}}$ and $v_{k,\text{prev}}$ that are merged at the loop's head should be the same; *ii*) correspondingly, ϕ-merged variable $v_{k,\text{next}}$ also has the same type T_k, a piece of information that is used to check the `while` block W and to type the condition itself c; *iii*) the loop body S is checked under ψ^+, which holds when c is true (staying condition); *iv*) conversely, the next block N is checked under the complementary ψ^- (exit condition); *v*) any return points in W, S, and N should return the same type T_{ret} of the whole loop; *vi*) finally, the whole instruction block always terminates the function if `while` block W or the `next` block always does (i.e., $\tau_W \vee \tau_N$).

Rule conditional works in a similar fashion: *i*) ψ^+, which holds when condition c is true, is used to check the S_{then} block, whereas the complement condition ψ^- is used to check the S_{else} block; *ii*) the type T_k of variables $v_{k,\text{then}}$ and $v_{k,\text{else}}$ that are merged after the conditional should be the same, and is also propagated to the ϕ-merged variable v_k; *iii*) typing the next block N can use condition ψ^+ iff τ_{else}, and condition ψ^- iff τ_{then}: in fact, if the `else` block terminates the enclosing function ($\tau_{\text{else}} = \top$), only the `then` condition ψ^+ holds after the conditional, and vice versa for ψ^- and τ_{then}; *iv*) finally, the whole instruction block always terminates the function (i.e., $\tau = \top$) if both the `then` and `else` always terminate (i.e., $\tau_{\text{then}} \wedge \tau_{\text{else}}$) or the `next` block always does (i.e., $\tau_{\text{after}} = \top$).

Expressions. Fig. 4b presents the main typing rules for expressions, most of which are straightforward: *i*) subtype and var formalize the usual subtyping and typing relations; *ii*) call models a qualified call, where the callee `f`'s signature is resolved based on the target v_0's type; *iii*) int-literal, bool-literal, and unit-literal are the obvious typing rules for

function

$$\frac{\Gamma, \langle v_k : T_k \rangle^k \vdash S \Uparrow T_{\text{ret}}, \top}{\Gamma \vdash \texttt{fn}\ T_0.\texttt{fun}(v_1 : T_1, \ldots) \to T_{\text{ret}}\ \{\ S\ \}}$$

assignment

$$\frac{\Gamma \vdash e : T; \psi^+ | \psi^- \qquad \Gamma, v : T; \psi^+ | \psi^- \vdash S \Uparrow T_{\text{ret}}, \tau}{\Gamma \vdash v := e\ ;\ S \Uparrow T_{\text{ret}}, \tau}$$

empty

$$\vdash \epsilon \Uparrow \texttt{Nothing}, \bot$$

ascription

$$\frac{\Gamma \vdash v : V \qquad \Gamma, v : V \vdash S \Uparrow T_{\text{ret}}}{\Gamma \vdash v :: V\ ;\ S \Uparrow T_{\text{ret}}}$$

return

$$\frac{\Gamma \vdash v : T}{\Gamma \vdash \texttt{return}\ v \Uparrow T, \top}$$

panic

$$\frac{\Gamma \vdash v : \texttt{String}}{\Gamma \vdash \texttt{panic}\ v \Uparrow \texttt{Nothing}, \top}$$

loop

$$\frac{\begin{array}{c}\langle \Gamma \vdash v_{k,\text{init}} : T_k \rangle^k \\ \langle \Gamma \vdash v_{k,\text{prev}} : T_k \rangle^k \qquad \Gamma, \langle v_{k,\text{next}} : T_k \rangle^k \vdash W \Uparrow T_{\text{ret}}, \tau_W \qquad \Gamma, \langle v_{k,\text{next}} : T_k \rangle^k \vdash c : \texttt{Bool}; \psi^+ | \psi^- \\ \Gamma, \psi^+, \langle v_{k,\text{next}} : T_k \rangle^k \vdash S \Uparrow T_{\text{ret}}, \tau_S \qquad \Gamma, \psi^-, \langle v_{k,\text{next}} : T_k \rangle^k \vdash N \Uparrow T_{\text{ret}}, \tau_N\end{array}}{\texttt{from}\ \langle v_{k,\text{next}} := \phi(v_{k,\text{init}}, v_{k,\text{prev}}) \rangle^k\ \texttt{while}\ W(c)\ \texttt{loop}\ S\ \texttt{next}\ N\ \Uparrow T_{\text{ret}}, \tau_W \vee \tau_N}$$

conditional

$$\frac{\begin{array}{c}\Gamma \vdash c : \texttt{Bool}; \psi^+ | \psi^- \qquad \Gamma, \psi^+ \vdash S_{\text{then}} \Uparrow T_{\text{ret}}, \tau_{\text{then}} \qquad \Gamma, \psi^- \vdash S_{\text{else}} \Uparrow T_{\text{ret}}, \tau_{\text{else}} \qquad \langle \Gamma \vdash v_{k,\text{then}} : T_k \rangle^k \\ \langle \Gamma \vdash v_{k,\text{else}} : T_k \rangle^k \qquad \Gamma, \langle v_k : T_k \rangle^k, \{\psi^+ \mid \tau_{\text{else}}\}, \{\psi^- \mid \tau_{\text{then}}\} \vdash N \Uparrow T_{\text{ret}}, \tau_{\text{after}} \qquad \tau = (\tau_{\text{then}} \wedge \tau_{\text{else}}) \vee \tau_{\text{after}}\end{array}}{\Gamma \vdash \texttt{if}\ c\ \texttt{then}\ S_{\text{then}}\ \texttt{else}\ S_{\text{else}}\ \texttt{merge}\ \langle v_k := \phi(v_{k,\text{then}}, v_{k,\text{else}}) \rangle^k\ \texttt{next}\ N \Uparrow T_{\text{ret}}, \tau}$$

(a) Instructions and control flow.

subtype

$$\frac{\Gamma \vdash v : T_1 \qquad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash v : T_2}$$

call

$$\frac{\Gamma \vdash \langle v_k : T_k \rangle^k \qquad \texttt{f} : \langle U_k \rangle^k \to T_{\text{ret}} \qquad \langle T_k <: U_k \rangle^k}{\Gamma \vdash \texttt{call}\ v_0.\texttt{f}(v_1, \ldots) : T_{\text{ret}}}$$

var

$$\frac{v : T \in \Gamma}{\Gamma \vdash v : T}$$

hybrid-cast

$$\frac{\Gamma \vdash v : B}{\Gamma \vdash v\,!! : B\ \texttt{with}\ p \wr p(v)}$$

cast

$$\frac{\Gamma \vdash v : B_1 \qquad \Gamma \vdash B_2 <: B_1}{\Gamma \vdash v\ \texttt{as}\ B_2 : B_2}$$

and

$$\frac{\Gamma \vdash b_1 : \texttt{Bool}; \psi_1^+ | \psi_1^- \qquad \Gamma \vdash b_2 : \texttt{Bool}; \psi_2^+ | \psi_2^-}{\Gamma \vdash b_1 \wedge b_2 : \texttt{Bool}\ ;\ \psi_1^+, \psi_2^+ \mid \emptyset}$$

or

$$\frac{\Gamma \vdash b_1 : \texttt{Bool}; \psi_1^+ | \psi_1^- \qquad \Gamma \vdash b_2 : \texttt{Bool}; \psi_2^+ | \psi_2^-}{\Gamma \vdash b_1 \vee b_2 : \texttt{Bool}\ ;\ \emptyset \mid \psi_1^-, \psi_2^-}$$

not

$$\frac{\Gamma \vdash b : \texttt{Bool}; \psi^+ | \psi^-}{\Gamma \vdash !b : \texttt{Bool}; \psi^- | \psi^+}$$

type-test

$$\frac{\Gamma \vdash b : B}{\Gamma \vdash b\ \texttt{is}\ B : \texttt{Bool}\ ;\ b : B \mid \emptyset}$$

comparison

$$\frac{\Gamma \vdash a_1 : \texttt{Int} \qquad \Gamma \vdash a_2 : \texttt{Int}}{\Gamma \vdash a_1 \bowtie a_2 : \texttt{Bool}\ ;\ a_1 \bowtie a_2 \mid \neg(a_1 \bowtie a_2)}$$

int-literal

$$\frac{n \in \mathbb{Z}}{n : \{n\}}$$

bool-literal

$$\frac{b \in \{\texttt{true}, \texttt{false}\}}{\vdash b : \texttt{Bool}}$$

unit-literal

$$\vdash \texttt{unit} : \texttt{Unit}$$

arith

$$\frac{\Gamma \vdash a_1 : I_1 <: \texttt{Int} \qquad \Gamma \vdash a_2 : I_2 <: \texttt{Int}}{\Gamma \vdash a_1 \boxdot a_2 : I_1 \odot I_2}$$

(b) Expressions.

refinement

$$\frac{\Gamma \vdash v : B \qquad \Gamma \vdash p(v)}{\Gamma \vdash v : B\ \texttt{with}\ p(\texttt{it})}$$

sub-ref

$$\frac{\Gamma \vdash B_1 <: B_2 \qquad \Gamma \vdash q_1 \Rightarrow q_2}{\Gamma \vdash B_1\ \texttt{with}\ q_1 <: B_2\ \texttt{with}\ q_2}$$

mono

$$\frac{\Gamma \vdash v : I <: \texttt{Int} \qquad M(v) = \texttt{[}\, a\,..\,b\,\texttt{]}}{\Gamma \vdash v : I \sqcap M(v)}$$

(c) Subtyping relation for refinement types, and usage rules for the monotonicity analysis.

Fig. 4: RANGER's typing rules.

literal of different types. The other expression rules are specialized to match RANGER's flow-sensitivity: *i*) and, or, and not augment the usual rules for Boolean expressions with the propagation of the flow conditions ψ^+, ψ^-; for example, and shows that the condition that holds when $b_1 \wedge b_2$ is true is the union of ψ_1^+ and ψ_2^+, since both b_1 and b_2 are true; *ii*) similarly, type-test and comparison augment the usual rules for type tests and comparison expressions with an encoding of their flow conditions; *iii*) arith captures a simple form of abstract interpretation over the interval domain [8], which derives a sound approximation of the range of an expression from its operands' ranges: $\odot$ is an abstract version of the arithmetic operations over this abstract domain; for example, `[`a`..`b`]` $\oplus$ `[`c`..`d`]` is range `[`$a + c$`..`$b + d$`]`.

Casts and hybrid casts. Fig. 4b also displays the typing rules for casts. Rule cast models a normal cast that narrows the type of v to a *base* subtype of its currently known type. In contrast, rule hybrid formalizes the hybrid cast operator `!!`, which provides an unsound escape hatch for the static typing rules. The rule allows the type checker to *assume* that v`!!` has refinement type B `with` p`(it)` but only checks statically that v has base type B; the check that $p(v)$ holds is deferred to run time (denoted $\wr\, p(v)$ in the rule). The term "hybrid cast" is taken from [10], where it denotes a similar feature.

Refinements rules. Fig. 4c presents the subtyping rules related to refinement types. Rule refinement encodes the semantics of the `with` refinement construct. Then, rule sub-ref reduces the subtyping relation between two refinement types to subtyping between their base types *and* implication between their refinement predicates. In particular, rule refinement implies that any type T is equivalent to T `with true`; and rule sub-ref—evaluated with $q = q_1$, $B = B_1 = B_2$, and q_2 = `true`—implies that B `with` $q <: B$ for any predicate q. We omit RANGER's other subtyping rules (esp. those for union and intersection types), as they are very similar to those of well-known type systems such as Scala's.

Monotonicity. Rule mono in Fig. 4c illustrates how RANGER's type checking uses the monotonicity information collected in a previous analysis. For every integer loop variable v that has been shown to be monotonically nondecreasing (range `[a..]`), nonincreasing (range `[..b]`, or constant (range `[a..a]`), RANGER narrows the type of v by intersecting it with the corresponding range type.

Using type candidates to help typechecking. Following the *bidirectional* typechecking paradigm [9], RANGER uses the type candidates CND as *heuristics* to help type analysis in two phases. First, it uses type candidates to sift through the predicates introduced by smart casting: in fact, smart casting predicates reflect a program's path conditions, and hence they can become quite complex; as a result, using all predicates to refine integer-typed expressions may proliferate the number of implications to check, which can slow down type checking considerably. Instead, RANGER's type checking algorithm uses a few heuristics to select which smart casting predicates to use; in particular, it prefers predicates that determine subtypes of any of the candidate types. Second, when RANGER's type checking algorithm processes a loop, it needs to guess the types T_k of the variables v_k merged in the `from` ϕ-function block; in particular, the type of the $v_{k,\text{prev}}$ variables is unknown initially, since type checking is a forward analysis. To this end, RANGER selects one of the available type candidates $C \in \text{CND}(v_{k,\text{prev}})$ at the loop's entry, checks that C is compatible with the type of the other merged variable $v_{k,\text{init}}$, and provisionally assumes that $v_{k,\text{prev}} : C$; when reaching the loop body's end, it will have enough information

about $v_{k,\text{prev}}$ to finally validate the assumption. If validation fails, typechecking will also fail: the inferred information was insufficient, and hence additional annotations are required. In all, the type checking algorithm ensures soundness, so that type inference only provides best effort suggestions.

Example. In Fig. 3's example, RANGER first applies rules int-literal and assignment to determine k_0, $max_0 : \{0\}$. Rule loop uses the loop condition to smartcast k_1 to `[..<a.length()]`, while monotonicity analysis strengthens k_1's type to `[0..<a.length()]`, ruling out out-of-range errors in the array access `a.at(k1)`. Furthermore, when starting to analyze the loop, RANGER guesses the type `[0 ..]` for max_1 (return type back-propagated as a type candidate). Thanks to rule conditional and the condition that guards the assignment of max_2, RANGER derives $max_2 :$ `[1 ..]` and $max_3 :$ `[0 ..]`, the latter validating the guess.

Purity. Predicates appearing in refinement types must be *pure* functions. RANGER includes a sound purity analysis for methods and closures, which also works at the level of IRCORNE. In short, a **`pure`** function can only call other pure functions, and compute and return the value of pure expressions—possibly split into intermediate assignments.

Other features. For space constraints, we cannot discuss in detail how RANGER is not limited to basic imperative constructs but integrates well with LICORNE's object-oriented and functional features, including classes, enum-like datatypes, and constrained genericity. For example, the return type of Fig. 1's function `filter` depends on argument `p`, used as a predicate to refine a type parameter. RANGER also (soundly) relaxes LICORNE's method overriding rules: while a method's argument types must be invariant in LICORNE (like in Java or Kotlin), RANGER allows an overriding redefinition of an argument of type τ **`with`** p to be invariant in τ but contravariant in p.

4.4 Implementation

We implemented RANGER on top of the LICORNE experimental programming language. The implementation language of both LICORNE's compiler and RANGER's type-checker is Scala. In the current prototype, the type checker implementation is approximately 4400 LOC, whereas the AST-to-IR conversion is about 1300 LOC. Fig. 5 shows the typechecking workflow, whose individual phases we described in Sec. 4.3. It runs on an intermediate representation that is a more complex version of Fig. 2's IRCORNE. As seen in the previous sections, RANGER's bidirectional type checking algorithm heavily relies on control-flow information that is more readily available in a bespoke SSA-form IR. This is why RANGER's implementation performs type checking at the IR level—in contrast to most mainstream programming languages that perform typechecking on the AST.

Constraint solving. The type checker's implementation calls out to the Z3 SMT solver [24] to check type constraints involving ranges and other refinement types: *i*) the subtyping relation between refinements involves checking the validity of an implication (rule sub-ref in Fig. 4c); *ii*) monotonicity analysis checks the validity of inequalities involving integer variables updated in a loop; *iii*) a type simplification mechanism that tries to make error messages more readable relies on SMT solving to rewrite range bounds in succinct form; *iv*) the simple form of abstract interpretation performed by Fig. 4b's rule arith also uses SMT solving.

In general, the constraints sent to the SMT solver are in the quantifier free fragment of integer arithmetic and uninterpreted functions theories [18], which is undecidable (recursively enumerable). When type annotations only involve range types (not general

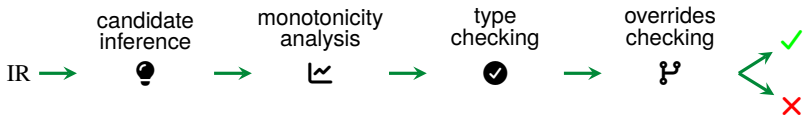


Fig. 5: An overview of RANGER's typechecking workflow.

refinements) and the corresponding numeric variables are only linearly updated, then the constraints are in integer *linear* arithmetic, which is NP complete. Despite these worst-case bounds, as confirmed in Sec. 5's experiments, RANGER's typechecking is fast on "natural" programs. In addition, RANGER's implementation reduces the complexity of the constraints that it reasons about in different ways: *i*) The candidate type inference's dataflow equations (Tab. 1) are not iterated to fixpoint but only unrolled once per loop; while this may produce unsound inference, this is not a problem because the typechecking phase validates any candidates before using them. *ii*) The type checker uses e-graphs [35] to transparently propagate typing information to all variables that are equal at a certain program point. This leverages the flow-sensitive information to avoid applying the same typing judgments redundantly.

5 Evaluation

The experimental evaluation targets two research questions:

RQ1 Does RANGER require a similar number of annotations as standard type systems?

RQ2 How does RANGER compare to similar frameworks in expressiveness, conciseness, soundness, and precision?

5.1 Experimental Setup

Comparable systems. LICORNE's features are closest to Java, Kotlin, and Scala. Since the latter has the most advanced type inference, we compare annotation overhead in Scala and RANGER to investigate RQ1. For RQ2, we compare RANGER to type systems for similar languages that can express ranges and whose implementation is available and usable. Thus, we select the Checker Framework [25] (especially its Index Checker [17]) and Liquid Java [14] as comparable systems. In contrast, we exclude other refinement type systems mentioned in Sec. 2 because they extend languages with fundamentally different type systems (e.g., Haskell, TypeScript, Rust) or their implementations are unmaintained (e.g., qualified types for Scala [30]).

Subjects. We collected a total of 14 example programs in 5 groups—listed in Tab. 2—from various sources that showcase different usages of range types. We wrote the programs in group rng ourselves, expressly to demonstrate RANGER's capabilities: these include data structures that represent dates and times (DateTime), integer filtering functions (FilterLess), and Fig. 3's and Fig. 1 motivating examples (MaxPos and Decoder). Group amap is a simplified version of the array map implementation in the Plume graph library [27], which we targeted because it already uses developer-written Checker Framework annotations. Group sort is a merge-sort implementation from [31] annotated for the Checker Framework by a student of the *Software Analysis* course [13]; we revised their annotations to make them as concise as possible. Subjects in group ic are from the Checker Framework manual [6, § 11], where they demonstrate using the Index Checker to verify correct index usage in array-manipulating programs. Programs in group liq are from various LiquidJava resources [21,20,22]. Groups liq and ic collect all meaningful examples we could find

in the Index Checker's and Liquid Java's official documentation and repositories that feature range refinement types. Several of the examples deliberately include variants with errors to also validate the soundness of the tools.

Setup. We first encoded each subject in Licorne, Java, and Scala with standard type annotations; the translations are behaviorally identical and, as much as possible, strike a balance between using each language's idiomatic features and retaining direct comparability. In these translations, the input and output parameters of each *unit* (a method or function, Tab. 2's column UN) is annotated with standard types: column STD reports the number of annotated parameters. Second, we refined the basic type annotations with range types wherever possible according to each framework's capabilities. Column REFIN counts the number of such annotations in signatures (a proxy for *conciseness*), and CONSTR the number of underlying constraints they express (a proxy for *expressiveness*). For example, the integer range from 0 to n included is expressible as `[0..n]` in RANGER and as `@Nonnegative @LessThan("n + 1")` in the Index Checker; thus, CONSTR is 2 in both (lower and upper bounds), whereas REFIN is 1 in RANGER and 2 in the Index Checker. Third, we added refinement annotations in the body of the functions as required by each framework in order to successfully typecheck the function (or, if it contains a bug, to locate it); these AUXiliary annotations measure the user's annotation burden. Sometimes, typechecking only succeeds if we add explicit casts or other forms of forced assumptions; we also recorded the number of such CASts (except for LiquidJava, which provides no such casting mechanism). Finally, we ran each framework's typechecking on the examples, and recorded the outcome for each unit: 1. a true positive TP is when a buggy unit fails typechecking, and hence the type checker finds the bug; 2. a false positive FP is when a correct unit fails typechecking (even if we add explicit casts), indicating a completeness failure in the type checker; 3. a true negative TN is when a correct unit typechecks successfully; 4. a false negative FN is when a buggy unit typechecks, indicating a soundness failure in the type checker. To enable a meaningful comparison between different tools that may report multiple errors differently, we count a unit as buggy (column B) iff it contains at least one bug. While working on these experiments, the LiquidJava verifier crashed on several examples (indicated as – in Tab. 2); given these issues, we did not include the two more complex subjects in LiquidJava's comparison (empty cells in Tab. 2).[3]

5.2 Experimental Results

RQ1. To answer RQ1, we compare columns AUX in Tab. 2, which report the number of auxiliary, in-body type annotations required in the examples. RANGER (column R) requires no auxiliary annotations, whereas Scala (column S) requires five. This clearly indicates that RANGER's expressive features do not introduce a heavier annotation burden than a standard language's type system. In fact, RANGER uses no auxiliary annotations at all, whereas Scala does in two examples where initializing a generic array or collection requires explicit type parameter annotations. One such example is program ArrayWrap, where Scala's behavior is possibly the result of using a `ClassTag` object. The other is program Decoder, where there are four occurrences of the common pattern: instantiate a collection, update it in a loop, and return it. Here, RANGER's type inference algorithm

[3] While preparing the examples, we contacted the authors of LiquidJava and revised our examples incorporating their suggestions on how to use their tool.

						LOC		REFIN			CONSTR			AUX				CAS		TP			FP			TN			FN		
GROUP	PROG	UN	B	STD	CSR	R	S	R	C	L	R	C	L	R	C	L	S	R	C	R	C	L	R	C	L	R	C	L	R	C	L
rng	DateTime	14	2	30	20	65	61	10	10	10	20	19	20	0	0	0	0	0	0	2	2	0	0	0	0	12	12	12	0	0	2
	FilterLess	4	2	12	4	24	14	4	4		4	4		0	2		0	0	0	2	1		0	1		2	1		0	1	
	MaxPos	4	2	14	11	56	56	10	9	–	11	11	–	0	0	–	0	0	0	2	2	–	0	0	–	2	2	–	0	0	–
	Decoder	8	0	23	17	44	52	12	15	–	17	15	–	0	4	–	4	0	5	0	0	–	0	1	–	8	7	–	0	0	–
amap	ArrayMap	8	1	15	9	57	54	6	6		9	9		0	0		0	0	0	1	1		0	0		7	7		0	0	
sort	Sorting	3	1	13	21	37	35	8	6	5	21	21	21	0	1	0	0	1	3	1	1	0	0	0	0	2	2	2	0	0	1
ic	ThirdElem	4	2	8	4	12	12	4	4	4	4	4	4	0	0	0	0	0	0	2	2	0	0	0	0	2	2	2	0	0	2
	ArrayLess	2	1	6	2	20	20	2	2	2	2	2	2	0	0	0	0	0	0	1	1	0	0	0	0	1	1	1	0	0	1
	RemString	3	1	9	0	27	20	0	0	–	0	0	–	0	0	–	0	0	0	1	1	–	0	0	–	2	2	–	0	0	–
	ArrayWrap	6	0	10	8	19	13	5	6	–	8	8	–	0	0	–	1	0	0	0	0	–	0	0	–	6	6	–	0	0	–
lj	Fibonacci	2	2	4	6	14	14	6	4	4	6	4	6	0	0	0	0	0	0	2	0	1	0	0	0	0	2	0	0	0	1
	Sum	2	1	4	4	18	18	4	2	2	4	2	4	0	0	0	0	0	0	1	0	1	0	0	0	1	2	1	0	0	0
	AbsDiv	3	1	7	4	9	9	4	3	4	4	2	4	0	0	0	0	0	0	1	0	0	0	0	0	2	3	2	0	0	1
	Car	5	1	4	7	22	22	4	4	4	7	7	7	0	0	0	0	0	0	1	1	1	0	0	0	4	4	4	0	0	0
	Total	68	17	159	117	424	400	79	75	35	117	108	68	0	7	0	5	1	8	17	12	3	0	2	0	51	53	24	0	1	8

Table 2: Evaluation results. Each row refers to a PROGram within a GROUP that implements a number of code UNits (methods/functions). For each program, the table reports the number B of units with bugs, the number STD of standard types in signatures (typed parameters plus return type), the maximal number CSR of range-like refinement constraints. The rest of the table reports data for each framework or language: LICORNE with RANGER annotations (R), Java with Checker Framework (C) and with LiquidJava (L) annotations, and regular Scala (S). These data include: the number LOC of lines of code of each unit, the number of individual REFINement annotations in signatures (a `@` annotation for C and L, and a range or `with` for R), the number of CONSTRaints these refinements express, the number of AUXiliary annotations inside the function body, the number CAS of forced casts (hybrid casts in R, `@AssumeAssertion` in C), the number of units that are true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). Dashes – denote cases where the tool crashed.

backpropagates the return type to the instantiation point, thus removing the need for an explicit annotation. Interestingly, in these examples Scala can perform *forward* inference instead, so that one can omit the return type in the signature. While we may argue that an explicitly typed signature is more generally useful than an in-body annotation, it remains clear that using RANGER's type system is no more burdensome or verbose than using a mainstream language like Scala, while providing additional static correctness guarantees.

RQ2. Let's compare RANGER (columns R in Tab. 2) to the Checker Framework and LiquidJava (columns C and L) for expressiveness, soundness, precision, and conciseness.

Expressiveness is measured by the fraction CONSTR/CSR of constraints expressible in each framework. According to this metric, RANGER is the most expressive (100% of constraints are expressible) whereas the Checker Framework can express 92% of constraints. LiquidJava also reaches 100% expressiveness on all the examples that we could tackle, but its expressiveness is undefined on the examples where it crashes. In all, these results demonstrate the higher expressiveness of systems based on free refinement types—as opposed to the fixed set of predefined annotations offered by the Checker Framework: the Constant Value Checker and Index Checker annotations can only encode constant variable bounds and the most common collection access patterns. For instance, the only lower bounds that the Index Checker can express are -1, 0, and 1.

Conciseness is measured as (REFIN + AUX + CAS)/CONSTR, namely the average number of all annotations needed to encode the expressible range constraints: the smaller, the more concise. RANGER uses 0.68 annotations per constraint, which is significantly more concise than the Checker Framework's 0.83 annotations per constraint; with its 0.51 annotations per constraint, LiquidJava is the most concise—with the important caveat that the practical significance of this result is limited by its incompatibility with certain Java

features. As for expressiveness, free refinement type notations tend to be more succinct than those using a predefined set; while we did not rigorously evaluate readability, it is plausible that conciseness is beneficial to it as well.

Soundness is measured by the fraction $\text{TN}/(\text{TN} + \text{FN})$ of all correct units that pass typechecking. RANGER did not incur any false negatives in our experiments; hence, its soundness is 100%. The Checker Framework incurred one FN in FilterLess, where it failed to detect an off-by-one bug when a filter function was applied to a stream; indeed, we have encountered other examples that suggest that the Checker Framework's support for the `Stream` class is sometimes limited. LiquidJava, in contrast, presented 7 false negatives, which gives a soundness of 76%. These failures usually involve type arguments, which seems to indicate that LiquidJava does not soundly model generics.

Precision is measured by the fraction $\text{TP}/(\text{TP} + \text{FP})$ of all found bugs (i.e., buggy units that fail typechecking). In our experiments, RANGER achieved 100% precision, whereas the Checker Framework was 86% precise. LiquidJava also reached 100% precision on all the examples that we could tackle; however, this result should be weighted against LiquidJava's significant number of soundness failures. The Checker Framework's precision loss comes down to its limited constraint reasoning capabilities, in contrast to RANGER's flexible usage of an SMT solver. A glaring example of the former's limitation occurs in Fig. 1's example, where it cannot verify without a forced assumption that the second argument in the call to `clamp` in `decode` is nonnegative: its solver cannot check the equality `xLen - 1 + 1 = xLen`. Another example is when instantiating two arrays `a`, `b` with the same length `s`: the Checker Framework only recognizes this fact if `b` is initialized as `new int[a.length]`, but not if as `new int[s]`; RANGER supports both patterns as equivalent.

Performance. While a thorough benchmarking of RANGER's performance is outside the scope of this paper, we collected basic performance data to ascertain that its typechecking algorithm is reasonably fast in practice. In our experiments, RANGER typechecked all the 14 examples in Tab. 2 in 16s; by comparison, the Scala 3 compiler took 34s to typecheck the same examples translated to Scala (note that we only measured the *typechecking* time, not the whole compilation time). These results indicate that RANGER's typechecking process is practically usable, and that its underlying program analyses are generally efficient.

6 Discussion and Conclusions

RANGER is an approach to range refinement types that focuses on ease of use and light annotation overhead rather than maximal expressiveness; in other words, it tries to extend the capabilities of standard type systems while retaining their ergonomics. This led to a type system that supports features like flow sensitivity with a custom backward inference algorithm, which significantly lowers the user annotation burden while naturally capturing common programming idioms. In our experiments, RANGER used annotations in measure comparable to a standard language's type system; and provided more expressivity, conciseness, or capabilities than comparable range refinement type systems. These results also showcase the advantages of weaving the design of a refinement type systems organically into a language's idioms—as opposed to retrofitting it ex post. In future work, we plan to generalize our approach to other kinds of refinement types and try to implement it for existing programming languages.

References

1. Aebi, V.: Implementation of a gradually compartmentalized programming language. Master's thesis, EPFL, Lausanne, Switzerland (January 2025), available at https://valentinaebi.github.io/resources/ms-thesis.pdf
2. Appel, A.W.: SSA is functional programming. ACM SIGPLAN Notices **33**(4), 17–20 (1998). https://doi.org/10.1145/278283.278285
3. Brady, E.: Idris, a general-purpose dependently typed programming language: Design and implementation. J. Funct. Program. **23**(5), 552–593 (2013). https://doi.org/10.1017/S095679681300018X
4. de Bruijn, N.G.: The mathematical language AUTOMATH, its usage, and some of its extensions. In: Symposium on Automatic Demonstration. Lecture Notes in Mathematics, vol. 125, pp. 29–61. Springer (1970). https://doi.org/10.1007/BFb0060623
5. Callaú, O., Robbes, R., Tanter, E., Röthlisberger, D., Bergel, A.: On the use of type predicates in object-oriented software: The case of Smalltalk. In: Proc. 10th ACM Symposium on Dynamic Languages. pp. 135–146. ACM (2014). https://doi.org/10.1145/2661088.2661091
6. The Checker Framework manual: Custom pluggable types for Java. https://checkerframework.org/manual/, last accessed 2026/06/27
7. Coquand, T., Huet, G.P.: The calculus of constructions. Inf. Comput. **76**(2/3), 95–120 (1988). https://doi.org/10.1016/0890-5401(88)90005-3
8. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 238–252. ACM (1977). https://doi.org/10.1145/512950.512973
9. Dunfield, J., Krishnaswami, N.: Bidirectional typing. ACM Comput. Surv. **54**(5), 98:1–98:38 (jun 2021). https://doi.org/10.1145/3450952
10. Flanagan, C.: Hybrid type checking. In: Proc. 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006. pp. 245–256. ACM (2006). https://doi.org/10.1145/1111037.1111059
11. Fonseca, A., Santos, P., Silva, S.: The usability argument for refinement typed genetic programming. In: International Conference on Parallel Problem Solving from Nature. pp. 18–32. Springer (2020)
12. Freeman, T., Pfenning, F.: Refinement types for ML. SIGPLAN Not. **26**(6), 268–277 (jun 1991). https://doi.org/10.1145/113446.113468
13. Furia, C.A.: Software analysis course at USI: course material. https://github.com/bugcounting/software-analysis/
14. Gamboa, C., Canelas, P., Timperley, C., Fonseca, A.: Usability-oriented design of liquid types for Java. In: Proc. 45th International Conference on Software Engineering (ICSE '23). pp. 1520–1532. IEEE Press (2023). https://doi.org/10.1109/ICSE48619.2023.00132
15. Jones, S.P., Weirich, S., Eisenberg, R.A., Vytiniotis, D.: A reflection on types. In: Lindley, S., McBride, C., Trinder, P.W., Sannella, D. (eds.) A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday. pp. 292–317. Lecture Notes in Computer Science, Springer (2016). https://doi.org/10.1007/978-3-319-30936-1_16
16. Kazerounian, M., Vazou, N., Bourgerie, A., Foster, J.S., Torlak, E.: Refinement types for Ruby. In: International Conference on Verification, Model Checking, and Abstract Interpretation. pp. 269–290. Springer International Publishing, Cham (dec 2017)
17. Kellogg, M., Dort, V., Millstein, S., Ernst, M.D.: Lightweight verification of array indexing. In: Proc. 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018). pp. 3–14. Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3213846.3213849

18. Kroening, D., Strichman, O.: Decision Procedures: An Algorithmic Point of View. Springer, 2nd edn. (2016). https://doi.org/10.1007/978-3-662-50497-0
19. Lehmann, N., Geller, A.T., Vazou, N., Jhala, R.: Flux: Liquid types for Rust. Proc. ACM Program. Lang. **7**(PLDI), 169:1–169:25 (jun 2023). https://doi.org/10.1145/3591283
20. Liquidjava - Extending Java with Liquid Types, GitHub repository. https://github.com/liquid-java/liquidjava, last accessed 2026/06/21
21. LiquidJava examples, GitHub repository. https://github.com/liquid-java/liquidjava-examples, last accessed 2026/05/20
22. Open VSX page of the LiquidJava Visual Studio Code extension. https://open-vsx.org/extension/AlcidesFonseca/liquid-java, last accessed 2026/06/21
23. Martin-Löf, P.: Intuitionistic type theory, Studies in proof theory, vol. 1. Bibliopolis (1984)
24. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems. TACAS 2008. Lecture Notes in Computer Science, vol. 4963. Springer, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78800-3_24
25. Papi, M.M., Ali, M., Correa, T.L., Perkins, J.H., Ernst, M.D.: Practical pluggable types for Java. In: Proc. 2008 International Symposium on Software Testing and Analysis (ISSTA '08). pp. 201–212. Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1390630.1390656
26. Pierce, B.C.: Types and Programming Languages. The MIT Press (2002)
27. Plume documentation. https://github.com/plume-lib/plume-util, last accessed 2026/06/21
28. Rondon, P., Bakst, A., Kawaguchi, M., Jhala, R.: Csolve: Verifying C with liquid types. In: International Conference on Computer Aided Verification. pp. 744–750. Springer Berlin Heidelberg, Berlin, Heidelberg (jul 2012)
29. Rondon, P.M., Kawaguci, M., Jhala, R.: Liquid types. In: Proc. 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08). pp. 159–169. Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1375581.1375602
30. Schmid, G.S., Kuncak, V.: SMT-based checking of predicate-qualified types for Scala. In: Proc. 2016 7th ACM SIGPLAN Symposium on Scala (SCALA 2016). pp. 31–40. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2998392.2998398
31. The Algorithms - Java. https://github.com/TheAlgorithms/Java, last accessed 2026/06/21
32. Tobin-Hochstadt, S., Felleisen, M.: Logical types for untyped languages. In: Proc. 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010. pp. 117–128. ACM (2010). https://doi.org/10.1145/1863543.1863561
33. Vazou, N., Seidel, E.L., Jhala, R., Vytiniotis, D., Peyton-Jones, S.: Refinement types for Haskell. In: Proc. 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14). pp. 269–282. Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2628136.2628161
34. Vekris, P., Cosman, B., Jhala, R.: Refinement types for TypeScript. In: Proc. 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16). pp. 310–325. Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2908080.2908110
35. Willsey, M., Nandi, C., Wang, Y.R., Flatt, O., Tatlock, Z., Panchekha, P.: egg: Fast and extensible equality saturation. Proc. ACM Program. Lang. **5**(POPL), 1–29 (2021). https://doi.org/10.1145/3434304, https://doi.org/10.1145/3434304