Generic Refinement Types

NICO LEHMANN, University of California, San Diego, USA
COLE KURASHIGE, University of California, San Diego, USA
NIKHIL AKITI, University of California, San Diego, USA
NIROOP KRISHNAKUMAR, University of California, San Diego, USA
RANJIT JHALA, University of California, San Diego, USA

We present *Generic Refinement Types*: a way to write modular higher-order specifications that abstract invariants over function contracts, while preserving automatic SMT-decidable verification. We show how generic refinements let us write a variety of modular higher-order specifications, including specifications for Rust's traits which abstract over the concrete refinements that hold for different trait implementations. We formalize generic refinements in a core calculus and show how to synthesize the generic instantiations algorithmically at usage sites via a combination of syntactic unification and constraint solving. We give semantics to generic refinements via the intuition that they correspond to *ghost parameters*, and we formalize this intuition via a type-preserving translation into the polymorphic contract calculus to establish the soundness of generic refinements. Finally, we evaluate generic refinements by implementing them in Flux and using it for two case studies. First, we show how generic refinements let us write modular specifications for Rust's vector indexing API that lets us statically verify the bounds safety of a variety of vector-manipulating benchmarks from the literature. Second, we use generic refinements to refine Rust's Diesel ORM library to track the semantics of the database queries issued by client applications, and hence, statically enforce data-dependent access-control policies in several database-backed web applications.

CCS Concepts: • **Theory of computation** → **Type structures**; Logic and verification; • **Software and its engineering** → *Software verification.*

Additional Key Words and Phrases: Rust, liquid types, refinement types, polymorphism

ACM Reference Format:
Nico Lehmann, Cole Kurashige, Nikhil Akiti, Niroop Krishnakumar, and Ranjit Jhala. 2025. Generic Refinement Types. *Proc. ACM Program. Lang.* 9, POPL, Article 49 (January 2025), 29 pages. https://doi.org/10.1145/3704885

1 Introduction

Modularity is a wonderful principle that helps one read or write programs. However, it can be a real pebble in one's shoe when it comes to automatically verifying them. To enable reuse across multiple client with different contexts, modular specifications must *abstract over* concrete invariants that hold at those contexts. For example, consider a modular specification for a `HashMap` library that aims to statically ensure that every lookup uses a previously defined key. Such a library must export contracts for `insert`, `lookup` and `contains_key` methods that abstract over the predicate characterizing the map's `valid` keys:

Authors' Contact Information: Nico Lehmann, University of California, San Diego, San Diego, USA, nlehmann@ucsd.edu; Cole Kurashige, University of California, San Diego, San Diego, USA, ckurashige@ucsd.edu; Nikhil Akiti, University of California, San Diego, San Diego, USA, nakiti@ucsd.edu; Niroop Krishnakumar, University of California, San Diego, San Diego, USA, nkrishnakumar@ucsd.edu; Ranjit Jhala, University of California, San Diego, San Diego, USA, rjhala@ucsd.edu.

ACM 2475-1421/2025/1-ART49
https://doi.org/10.1145/3704885

```
pub struct HashMap<K, V>[valid: K -> bool] {...}

impl<K: Eq + Hash, V> HashMap<K, V> {
    pub fn new() -> Self[|k| false];

    pub fn lookup(&Self[@valid], key: K{ valid(key) }) -> &V;

    pub fn contains_key(&Self[@valid], key: K) -> bool[valid(key)];

    pub fn insert(self: &strg Self[@valid], key: K{ !valid(key) }, v: V)
    ensures self: Self[|k| valid(k) || k == key];
}
```

Sadly, the need to abstract invariants over contracts implicitly requires *higher-order* specifications which preclude decidable verification. For example, the contract for `contains_key` says that *for all* predicates `valid` that characterize the defined keys of the map `Self`, the output `bool` is equal to the value of the predicate for the input key. To verify a client, we must *instantiate* the predicate appropriately at the client call-sites, which is only feasible when working interactively with a proof assistant conversant in higher-order logic, and so, outside the realm of decidable verification.

Higher-Order Specifications via Generic Parameters In this paper, we show a way to reconcile modular higher-order specifications with automatic and SMT-decidable verification, by carefully combining two key observations. Our first observation, following Vazou et al. [28], is that the higher-order invariants can be viewed as *refinement parameters*, analogous to type parameters. Unfortunately, in the system of Vazou et al. [28], parameters are instantiated at call-sites via Horn constraint solving which comes with rather onerous terms and conditions: the refinement parameters may only appear *positively* inside refinements, *i.e.,* only under conjunctions, but *not* disjunctions or negations. This positivity prerequisite severely restricts the expressiveness of specifications. For example, we cannot write the contracts for `contains_key` or `insert` for the `HashMap` shown above! Following Economou et al. [10], we show how to circumvent this restriction by the second observation that base-sorted (*e.g.,* `int` or `bool` valued) refinement parameters can be automatically instantiated at usage sites via syntactic unification—regardless of whether they appear positively or negatively—as long as they appear in a *value-dependent position*. Our key insight is that the same syntactic unification technique can be generalized to instantiate higher-sorted (*e.g.,* function valued) refinement parameters, and doing so lets us write modular higher-order specifications, like for `HashMap` above, while preserving automatic verification.

In this paper, we combine these two observations to develop *Generic Refinement Types*, via the following concrete contributions.

1. Formalization First, we formalize generic refinement types in a core calculus λ_G, which extends the simply typed λ-calculus with generic refinement types (§ 4). Our calculus starts with an index-style formulation where base types (*e.g.,* `int`, `bool`) are *indexed* by logical values that track the exact value of the inhabitant [32, 33]. For example, the signature $\forall a : \mathbb{Z}.\ \texttt{int}[a] \rightarrow \texttt{bool}[a > 0]$ corresponds to a function that determines whether its input `int` is strictly positive using a as a *refinement parameter* that holds the logical value of the input integer. In λ_G, we extend (base-sorted) refinement parameters with two kinds of (function-sorted) generic arguments that permit quantification over invariants in contracts. The first kind, are *Horn* generics that get instantiated via Horn constraint solving but which must satisfy the positivity prerequisite [28]. The second kind, are *Hindley* generics that get instantiated via syntactic unification [10] as long as they appear in a value-dependent position. We develop the first algorithmic system for type checking programs with Horn and Hindley generics thus reconciling modular specification with decidable verification.

2. Translation Second, we give a semantics to generic refinements by showing how (well-typed) λ_G programs have a type-preserving translation into the polymorphic contract calculus $\mathrm{F}^{\sigma}_{\mathrm{H}}$ of Sekiyama et al. [24] (§ 5). Informally, generic refinements are viewed as *ghost* parameters that must be passed to functions, whose values are automatically synthesized during algorithmic type checking. We formalize this intuition by defining an intermediate calculus $\lambda_{\underline{G}}$ with explicit refinement instantiations which are elaborated by the algorithmic typing rules. We then prove that the elaboration is type-preserving, *i.e.,* that if the algorithmic system gives a term in λ_G a type, then there is a corresponding term in $\lambda_{\underline{G}}$ with the appropriately elaborated type (theorem 5.1). Finally, we show how the elaborated terms of $\lambda_{\underline{G}}$ can be translated to $\mathrm{F}^{\sigma}_{\mathrm{H}}$, again in a type-preserving fashion, which lets us use the soundness of $\mathrm{F}^{\sigma}_{\mathrm{H}}$ to establish the soundness of λ_G (theorem 5.2).

3. Association Rust libraries pervasively uses *traits* to abstract over concrete implementations of various groups of related operations, *e.g.,* accessing collections like vectors, slices, or (hash)maps, iteration, or building DSLs. We show how generic refinements enable modular specifications for trait methods, via *Associated generic refinements* which abstract over the specific concrete refinements that hold for particular implementations of the trait (§ 2). Following Wadler and Blott [30]'s classic observation that typeclasses can be translated to dictionaries, we illustrate how associated generics can be translated to Hindley generics, thereby showing how the latter is a foundation for the former.

4. Evaluation We evaluate generic refinements by implementing our core functional model, that focuses on Hindley parameters, as an extensionin to Flux, a refinement type verifier for Rust, and then carrying out two case studies (§ 6). In the first case study, we show how generic refinements scale up to modularly specify vector bounds checking. Their use allows us to abstract over the condition of when an index is in bounds into a generic refinement that is then implemented as the usual bounds-check for the vector. We show how this modular specification lets us statically verify the bounds safety of a variety of vector manipulating benchmarks from [16]. In the second case study, we use generic refinements to refine Rust's Diesel ORM library to track the semantics of the database queries issued by client applications, and hence, statically enforce data-dependent access-control policies in database-backed web applications [17].

2 Overview

We start with a high-level overview of generic refinements that begins by recapitulating prior work as *Horn* generic refinements (§ 2.1). Then, we describe the expressiveness limitations of Horn generics and show how they are addressed via *Hindley* generic refinements (§ 2.2). Next, we show how generic refinements scale up to permit modular specification and verification of code using Rust's traits via *Associated* generic refinements (§ 2.3). Finally, we illustrate how Hindley generics are a lower-level *foundation* for associated generic refinements, by demonstrating how the standard typeclasses-as-dictionaries translation [30] reduces associated generics to Hindley generics (§ 2.4).

2.1 Horn Generic Refinements

Vazou et al. [28] introduced a mechanism for abstracting or generalizing over refinements via type signatures with *refinement parameters* which can be automatically instantiated by solving Horn constraints. We recast this idea as *Horn* generic refinements, which we illustrate with an example.

Problem: Multiple Mutable Borrows Rust's borrowing discipline can sometimes be a straitjacket that makes certain programs rather uncomfortable to write, most notably, when we require *multiple* mutable references into objects stored in a collection. For example, suppose we represent particles using a type `Particle` that provides a method `interact`, which takes mutable references to two particles and updates them based on their interaction:

```
struct Particle {...};       impl Particle { fn interact(&mut self, other: &mut Self) }
```

```
fn sim_bad(vec: &mut Vec<Particle>) {
  let n = vec.len();
  for i in 0..n {

    let a = &mut vec[i];
    for j in i + 1..n {
      let b = &mut vec[j];
      a.interact(b);
    }
  }
}
```

```
fn sim(vec: &mut { Vec<Particle>[@n] | 0 < n }) {
  let n = vec.len();
  for i in 0..n {
    let mut mi = MultiIdx::new(vec);
    let a = mi.get(i);
    for j in i + 1..n {  // mi: MultiIdx<Particle>[p]
      let b = mi.get(j); // p = |k| 0<=k< i || j<=k<n
      a.interact(b);
    }
  }
}
```

Fig. 1. A `Particle` simulation that requires multiple `mut` borrows: the left version is rejected by Rust's borrow checker, but the right version is accepted.

We might now wish to implement a simulator that computes the interaction of a *collection* of `Vec<Particle>` as a simple nested loop as shown on the left in Fig. 1. Sadly, Rust's borrow checker will fuss about the second mutable borrow (at the `let b =`) as at that point we *already* have a mutable borrow `a` that is alive (as required at the call to `interact`) and so we *cannot* have a second mutable borrow from the same collection `vec`.

Solution: A Horn Generic API for Multiple Borrows Fig. 2 shows how Horn generics let us implement an API that allows for multiple mutable borrows while still ensuring the key *non-aliasing* invariant, namely, there is a unique borrow of any object. First, we define a `MultiIdx` struct that wraps the `Vec` and (temporarily) takes ownership of the underlying memory. Second, we parameterize `MultiIdx` with a *generic argument* `available: int -> bool` which is a predicate over `int` that describes the set of indices of the underlying vector that are *available* to be (mutably) borrowed. (We use the `hrn` modifier to declare the generic as Horn.) Third, our API ensures that the only way to *create* a `MultiIdx` is using the constructor `MultiIdx::new` which takes a vector `vec` and returns a `MultiIdx` whose `available` predicate says that *every* slot `i` satisfying `0 <= i < vec.len` is available to be mutably borrowed. Finally, to actually *borrow* a reference to an element in the collection, a client can use the method `get`. This method is parametric on the available predicate (declared with `@av`) and checks that the requested `idx` is indeed available, and then returns a mutable reference to the underlying cell after "updating" `self` to ensure that (subsequently) the position `idx` is *excluded* from the set of available borrows. The specification uses a strong reference, the details of which are not important. It suffices to say that, by leveraging Rust's borrow checker, Flux can do a sound strong update on the type of the reference after the function call [16].

Implementing this API requires internal use of `unsafe` code, but we can encapsulate it under a (refined) safe interface. The key observation that makes this sound is that by wrapping a mutable reference to the `Vec`, `MultiIdx` can prevent clients from using operations that can modify the underlying storage to invalidate outstanding borrows (e.g., by calling `push`).

Using the `MultiIdx` API The following example shows how we can use this API. The comments on the right illustrate how the type of `mi` is strongly updated after the first call to `get`. Flux rejects the second borrow at `0` as that index is *not* available after the first call `mi.get(0)`.

```
fn test_multi_index(vec: &mut Vec<Particle>{v: v.len == 5}) {
    let mi = MultiIdx::new(vec); // mi: MultiIdx[|i| 0 <= i < 5]
    let a0 = mi.get(0);          // mi: MultiIdx[|i| 0 <= i < 5 && i != 0]
    let a1 = mi.get(0);
}
```

```
struct MultiIdx<'a, T>[hrn available: int -> bool];

impl<'a, T> MultiIdx<'a, T> {
    fn new(vec: &'a mut Vec<T>]) -> Self[|i| 0 <= i && i < vec.len];

    fn get(self: &strg Self[@av], idx: usize{ av(idx) }) -> &'a mut T
    ensures self: Self[|i| av(i) && i != idx];
}
```

Fig. 2. An API for Multiple Mutable Borrows using a Horn generic refinement to track *available* positions.

A Verified Particle Simulator We can now write the `Particle` simulator quite naturally as shown in Fig. 1. Flux statically guarantees that each object is borrowed at most once, by automatically instantiating the `available` generic argument for `mi`, as shown on the right in the comments. In particular, Flux infers that the *inner loop* has the invariant that all the indices *excluding* those from `i` to `j` are `available`, thereby verifying that the `mb.get(j)` call can safely return a mutable borrow `b` that can then be used to call `interact`.

2.2 Hindley Generic Refinements

In general, generics are only ergonomic when they can be automatically instantiated [22]. In the case of Horn generics, automatic inference comes at a steep price: the refinement arguments (like `av` in the signature of `get` in Fig. 2) can only appear *positively* inside refinements, meaning they can only appear under conjunctions, but *not* disjunctions or negations. This *positivity prerequisite* lets the type checker automatically instantiate the generic arguments by solving Horn constraints [28], but severely limits the expressiveness of APIs that can by specified with Horn generics.

Next, let us see how to relax the posititivity prerequisite and expand expressiveness via *Hindley* generic refinements, which use *syntactic unification* to automatically instantiate generic arguments at usage sites. Hindley generic arguments can appear negatively—under a disjunction or negation—as long as they appear in a *value-dependent position* (a notion identified by Economou et al. [10], and detailed in § 3.2) which lets us use them to specify a broader class of APIs.

Querying Databases using ORMs An object-relational mapping (ORM) library let programmers interact with databases in a type safe manner using a high level API for constructing and executing database queries. An ORM starts off by letting the programmer describe the *schema* of their database tables. Suppose we are building a shopping list application that maintains the list of items each user needs to purchase, together with their price and description. The programmer describes the schema for a table as shown on the left in Fig. 3, after which the ORM library generates a `struct` to represent rows of the `items` table as a Rust value. An ORM provides a convenient way to build *queries* that can be used to read and update the database. To do so, the ORM uses a type `Field<R, V>` whose inhabitants represent the `V`-valued field of a row `R`. For example, from the `table` description on the left in Fig. 3, the ORM will automatically generate the definitions shown on the right which represent the database *columns*, *i.e.,* the fields of each row of the database. Next, as shown in Fig. 4, the ORM also has a type `Query<R>` which is parameterized by the type of the *row* `R` that the query returns, and a set of generic (table-independent) functions that allow the construction and execution of queries. For example, we can use `eq` or `lt` to build *atomic* queries that compare the value of a particular field or column (of the row) to some value `val`, and then, we can *compose* queries with boolean connectives like `and`, `or` and `not` to construct complex queries that span multiple fields. Finally, the ORM includes a method `run` that executes the query on the actual database, to get back a collection of rows of type `R`.

```
table! {
  items(id) {
    id    -> usize,
    price -> i32,
    owner -> usize,
    descr -> String,
  }
}
```

```
struct Item {
  id: usize,
  price: i32,
  owner: usize,
  descr: String,
}
```

```
const item_id    : Field<Item, usize>  = ... ;
const item_price : Field<Item, i32>    = ... ;
const item_owner : Field<Item, usize>  = ... ;
const item_descr : Field<Item, String> = ... ;
```

Fig. 3. (L) Specifying a database table using an ORM, (C) ORM-generated `struct` representing each row, (R) ORM-generated `Field`s representing each column.

```
impl<R, V> Field<R,V> {
  fn eq(self: Field<R,V>, val:V) -> Query<R>;
  fn lt(self: Field<R,V>, val:V) -> Query<R>;
  fn gt(self: Field<R,V>, val:V) -> Query<R>;
}
```

```
impl<R> Query<R> {
  fn and(self: Query<R>, q: Query<R>) -> Query<R>;
  fn or(self: Query<R>, q: Query<R>) -> Query<R>;
  fn not(self: Query<R>) -> Query<R>;
  fn run(self: Query<R>) -> Vec<R>;
}
```

Fig. 4. (L) Methods to build atomic queries from a field. (R) Methods to compose queries and run them.

ORM clients Client or application code can now use the ORM's API to build SQL queries and execute them to get back plain Rust values that can be consumed by the rest of the application. For example, Fig. 5 shows a Rust function that finds all the items belonging to `owner` whose price is below a certain `cost`. To do so, the code uses the query combinators to build a query `q` that is the conjunction of two sub-queries, which respectively check that the item belongs to `owner` and that the `price` is less than `cost`, and then `run`s the query to iterate over all the returned items.

Problem: Verifying Invariants of Query Results When verifying web-applications built using the ORM, we need to track the invariants of the rows returned by various queries, to check, *e.g.,* that the values being sent back to the user adhere to some desired security policy [17]. In our setting, suppose that we wish to verify that every `item` returned by `q.run()` indeed has a `price` that is less than some `cost`. To do so, we must precisely track the semantics of the query combinators like `eq`, `lt` and `and`, and compose them appropriately to track the desired invariants of `q` and hence `item`. However, for modularity, the types we assign to these combinators must be database agnostic so that we can use the same API across different applications and DB tables.

Solution: A Hindley-generic API for Database Queries We solve this problem by refining the ORM's `Field` and `Query` types with generic arguments that track the semantics of the respective objects, and then refine the API for constructing and executing a `Query` so that the refinements for the sub-queries *compose* to yield compile-time invariants for the resulting database rows.

Refined Fields First, we refine the `Field` struct with a Hindley generic (declared with the `hdl` modifier) `proj` which represents the map from the row `R` to the specific field's value `V`

```
struct Field<R, V>[hdl proj: R -> V] { ... }
```

Now, when the `table` from Fig. 3 is generated, the individual `Field` constants now get refined as

```
const item_id    : Field<Item, usize  >[|r| r.id   ] = ... ;
const item_price : Field<Item, i32    >[|r| r.price] = ... ;
const item_owner : Field<Item, usize  >[|r| r.owner] = ... ;
const item_descr : Field<Item, String >[|r| r.descr] = ... ;
```

```
fn get_items(owner: usize, cost: i32) {
    let q1 = item_owner.eq(owner);      // q1: Query<Item>[|i| i.owner == owner]
    let q2 = item_price.lt(cost);       // q2: Query<Item>[|i| i.price < cost]
    let q  = q1.and(q2);                // q: Query<Item>[|i| i.owner == owner && i.price < cost]
    for item in q.run() {               // item: Item{i: i.owner == owner && i.price < cost}
        assert(item.price < cost);
    }
}
```

Fig. 5. Example of a client of the query API. The comments show the types of subqueries in the refined API.

Refined Queries Next, we refine `Query` with a generic argument that characterizes its result.

```
struct Query<R>[hdl inv: R -> bool] { ... }

impl<R> Query<R>{ fn run(self: Query<R>[@q]) -> Vec<R{r: q(r)}>; }
```

Crucially, the `run` method guarantees that *each* row returned by `query.run()` will satisfy the invariant `q` that the `query` is refined with.

Atomic Queries We can now refine the types of the primitive query constructors to precisely track the semantics of the primitive queries

```
impl<R, V> Field<R, V> {
    fn eq(self: Field<R, V>[@proj], val: V) -> Query<R>[|r| proj(r) == v];
    fn lt(self: Field<R, V>[@proj], val: V) -> Query<R>[|r| proj(r) < v];
    fn gt(self: Field<R, V>[@proj], val: V) -> Query<R>[|r| proj(r) > v];
}
```

Thanks to the types of the fields `item_owner`, `item_price` and the query constructors `eq` and `gt` above, Flux assigns the sub-queries `q1` and `q2` in Fig. 5 the refined types shown in comments.

Composing Queries Next, we can refine the query combinator API to track the types of the constructed queries:

```
impl<R> Query<R> {
    fn and(self: Query<R>[@q1], rhs: Query<R>[@q2]) -> Query<R>[|r| q1(r) && q2(r)];
    fn or(self: Query<R>[@q1], rhs: Query<R>[@q2]) -> Query<R>[|r| q1(r) || q2(r)];
    fn not(self: Query<R>[@q]) -> Query<R>[|r| !q(r)];
}
```

Crucially, as they are no longer subject to the positivity prerequisite, the Hindley generic arguments can appear negatively under a disjunction or negation (as long as they also appear in a value-dependent position as detailed in § 3.2), which lets us use them to *compose* the invariants of sub-queries, to precisely track the invariants of the composed query.

Thus, using the type of `and` and the types of `q1` and `q2`, Flux infers that `q` (from Fig. 5) has the type shown next to it in comments. This lets Flux determine that `q.run()` returns a vector of items each of which satisfies the condition in the `assert`.

2.3 Associated Generic Refinements

Generic refinements scale up to provide a modular way to abstract specifications over Rust's widely used traits, and which mirrors Rust's own *associated types* mechanism.

Safe Vector Indexing Rust's `Vec<T>` is a workhorse type used ubiquitously to store and manipulate collections of objects. Vector-bounds safety is one of *the* oldest use cases for refinement types [32],

and so we might expect it would be straightforward to do vector-bounds checking with Flux, *e.g.*, to verify that the access `vec[i]` is safe in the following snippet, *i.e.,* `i` is between `0` and `vec.len()`.

```
fn sum(vec: &Vec<i32>) -> i32 {
    let mut res = 0; let mut i = 0;
    while i < vec.len() {
        res += vec[i];
        i += 1;
    }
    res
}
```

Trait-based Indexing Unfortunately, it is rather tricky to even *specify* vector access safety, because Rust uses *traits* to make the indexing operation `vec[i]` extensible. The standard library defines a trait `Index`—shown below on the left—that is parameterized by the type `Idx` used as the actual index.[1] The trait has an *associated type* `Output`, which represents the result of the indexing operation. Thus, the trait method `index` takes the value being indexed (`self`), the generic index (`idx`) and returns a reference to `Self::Output`. The actual `vec[i]` operation inside `sum` uses a particular *implementation* of the `Index` trait for `Vec<T>` using `usize` indices—shown below on the right—where the associated type `Output` is specialized to `T`.

```
trait Index<Idx> {
  type Output;
  fn index(&self, idx: Idx) -> &Self::Output;
}
```

```
impl<T> Index<usize> for Vec<T> {
  type Output = T;
  fn index(self: &Vec<T>, idx: usize) -> &T {...}
}
```

Problem: Specifying Safe Vector Indexing The indirection introduced by the trait leaves us with a conundrum: *where* do we specify that the index should be within bounds? One possibility is to refine the signature of the `index` method in the trait implementation block:

```
impl<T> Index<usize> for Vec<T> {
    fn index(self: &Vec<T>, idx: usize{idx < self.len}) -> &T { ... }
}
```

Now, the type checker could use the fact that the `vec[i]` operation in `sum` actually calls the above method to use the refined API and thus enforce index safety. However, this route is unsound in general as it is easy to bypass the check. For example, consider the below "wrapper" function to index any container `C` with a `usize`:

```
fn get<C: Index<usize>>(c: C, i: usize) -> &C::Output { c[i] }
```

The above definition of `get` would typecheck. However, we could completely bypass the (static) bounds check by replacing the `v[i]` in `sum` with `get(v,i)`! Thus, our only viable option is to somehow refine the definition of the `Index` trait. However, here we face a vexing puzzle. At the trait definition, the underlying container and index are both generic: we do not even know we are talking about vectors or `usize` indices, so how can we possibly refine the `index` method to require something so specific as an arithmetic bound?

Solution: Associated Generic Refinements We solve this conundrum by generalizing Horn- and Hindley- generics to the trait setting as *associated generic refinements*. At the trait-level, associated generic refinements abstract over different requirements for the different implementations. These generics can then be instantiated at the implementation-level, to the respective concrete requirements. Let us see how associated generics allow us to specify safe vector indexing.

[1]We present a slightly simplified account, the actual implementation uses *another* trait `SliceIndex` elided for clarity.

```
trait Index<Idx> {
    type Output;
    reft in_bounds(v: Self, idx: Idx) -> bool;
    fn index(self: &Self[@v], idx: Idx { Self::in_bounds(v, idx) }) -> &Self::Output;
}

impl<T> Index<usize> for Vec<T> {
    type Output = T;
    reft in_bounds(v: Self, idx: Idx) -> bool { idx < v.len }
    fn index(self: &Vec<T>, idx: usize) -> &T { ... }
}
```

Fig. 6. Specifying Safe Vector Indexing via an Associated Generic Refinement.

First, as shown at the top of Fig. 6, at the *definition* of `trait Index`, we extend Rust's notion of an associated (generic) type (*e.g.,* `Output`) to define an *associated (generic) refinement* `in_bounds` that is declared to be a binary predicate over `Self` (the generic container) and `idx` (the generic index). The associated generic `in_bounds` as a trait-level is an analog of the Horn generic `available` (§ 2.1), or Hindley generic `proj`, `inv`, `q1` and `q2` (§ 2.2). The concrete definition of `in_bounds` will be filled in at usage sites, in this case, the different implementations of `Index`. However, we can use the generic `in_bounds` in type specifications for the trait's methods, akin to the use of the associated type `Self::Output`. In particular, we can refine the type of the `index` method to require that the `idx` passed in be such that `Self::in_bounds(v, idx)` holds, *i.e.,* that the index `idx` is "within bounds" for the container `v`.

Second, as shown in the bottom of Fig. 6, at the implementation of `Index`—where we "know" that the indexing is specialized to `Vec<T>` with `usize` indices—we provide the actual concrete implementation of `in_bounds` (for this instance) which specifies that `idx < v.len`. Now, when type checking `sum`, Flux can use rustc's trait normalization machinery to instantiate the generic `in_bounds` in the precondition of `index` with the refinement in the actual implementation used at that call-site, thereby enforcing bounds safety!

Note that the associated generic eliminates the unsoundness introduced by wrappers like `get`, as the generic precondition on the trait's `index` method ensures that the definition of `get` does not typecheck as is. Instead, we would have to write

```
fn get<C: Index<usize>>(c: C, i: usize{ C::in_bounds(c, i) }) -> &C::Output { c[i] }
```

and the moment we do so, the occurrence of `get(v, i)` in `sum` would again instantiate the generic refinement `in_bounds` to the instance for `Vec` at that call-site, thus enforcing bound safety.

2.4 Associated Generics *are* Hindley Generics

We conclude the overview by sketching how the classic typeclasses-as-dictionaries translation [30] can be used to reduce Associated generics to Hindley generics, thereby showing the latter provides a solid foundation for the former. We illustrate the translation using the `Index` trait and the implementation for `Vec<i32>` as an example (Fig. 6).

1. Translate `trait` into `struct` First, we translate the `Index` trait into a *trait-struct* `IndexTrait` where the `trait`'s (1) associated types (*e.g.,* `Output`) become extra type parameters, (2) methods (*e.g.,* index) become `fn`-valued fields of the struct, and (3) associated refinements (*e.g.,* `in_bounds`) translate to a Hindley generic for the struct.

```
// trait Index translated into struct IndexTrait
struct IndexTrait<Self, Idx, Output>[hdl in_bounds: (Self, Idx) -> bool] {
    index: fn(self: &Self[@v], idx: Idx { in_bounds(v, idx) }) -> &Output;
}
```

2. Translate `impl` into instance Next, given the concrete (monomorphic) trait implementation of `Index<usize>` for `Vec<i32>`, we translate it into an *impl-instance* `impl_index_vec_i32` that is an instance of the corresponding trait-struct. The `impl`'s (1) associated types (*e.g.,* `Output`) are specialized to the concrete types (*e.g.,* `i32`), (2) method implementations (*e.g.,* `index`) are translated into concrete function values (*e.g.,* the actual lookup function), and crucially, (3) associated refinements (*e.g.,* `in_bounds`) are instantiated with the concrete refinement.

```
// impl Index<usize> for Vec<i32> translated as instance
let impl_index_vec_i32: IndexTrait<Vec<i32>, usize, i32>[|v,idx| idx < v.len] = IndexTrait {
    index: |self: &Vec<i32>, idx: usize{ idx < self.len }| -> &i32 { ... }
};
```

3. Translate `trait` bounds into explicit function parameters We translate functions that use a trait bound into functions taking an explicit parameter. This is illustrated below with the function `get` where the `C: Index<...>` bound (on the left) is translated into an explicit parameter `c_trait` of the correponding `IndexTrait` type instantiated with appropriate generics (on the right). Crucially, notice that the associated generic `in_bounds` for the trait bound translates into the Hindley generic for the trait-struct parameter. The actual use of the `index` method in the body is translated into a call to the `index` field of the `c_trait` struct, and the `in_bounds(c, i)` precondition on `i` translate to a call of the Hindley generic.

```
// get function with a trait bound
fn get<C: Index<usize>>(

    c: C,
    i: usize{ C::in_bounds(c, i) }
) -> &C::Output {
    c[i]
}
```

```
// get translated to use a trait-struct
fn get_tx<C, Output>(
    c_trait: &IndexTrait<C, usize, Output>[@in_bounds],
    c: &C,
    i: usize { in_bounds(c, i)}
) -> &Output {
    (c_trait.index)(c, i)
}
```

4. Translate bound instances by passing impl-instances Finally, we translate a call to `get`, like the one shown below on the left, into a version like the one on the right that explicitly passes `impl_index_vec_i32` as the argument to the translated trait-bound parameter `c_trait`.

```
// sum client that uses get
fn sum(vec: &Vec<i32>) -> i32 {
    let mut res = 0;
    let mut i = 0;
    while i < vec.len() {
        res += get(vec, i);
        i += 1;
    };
    res
}
```

```
// sum translated to use impl-instance
fn sum_tx(vec: &Vec<i32>) -> i32 {
    let mut res = 0;
    let mut i = 0;
    while i < vec.len() {
        res += get_tx(impl_index_vec_i32, vec, i);
        i += 1;
    };
    res
}
```

Base	σ_b	:=	$\mathbb{Z}$	*int*	***Mode***	μ	:=	**hdl**	*hindley*
		|	$\mathbb{B}$	*bool*			|	**hrn**	*horn*
Sort	σ	::=	σ_b	*base*	***Scheme***	η	:=	τ	*type*
		|	$\sigma_b \to \sigma_b$	*func.*			|	$\forall a{:}_\mu\, \sigma.\, \eta$	*generalization*
Reft.	r	::=	$z \mid \mathrm{tt} \mid \mathrm{ff} \mid a$	*atom*	***Expr.***	e	::=	x	*variable*
		|	$r \wedge r \mid r \vee r \mid \neg r$	*bool*			|	c	*constant*
		|	$r \bowtie r \mid r \oplus r$	*arith*			|	$\mathrm{op}(e_1, \ldots, e_n)$	*operation*
		|	$\lambda a : \sigma.\, r \mid r\ r$	*func.*			|	$\lambda x.\, e$	*lambda*
Type	τ	::=	$\{b[r] \mid r\}$	*refinement*			|	$e(e_1, \ldots, e_n)$	*application*
		|	$\{a.\, b[a] \mid r\}$	*existential*			|	$e : \tau$	*ascription*
		|	$\tau \to \tau$	*function*	***PCtx.***	Γ	:=	$\cdot \mid \Gamma, x : \tau$	
RCtx.	Θ	:=	$\cdot \mid \Theta, a{:}_\mu\, \sigma \mid \Theta, r$						

Fig. 7. Syntax of λ_G. An operation $\oplus$ is either + or −. A relation $\bowtie$ is either = or >.

At this point, Flux instantiates the Hindley generic for `get_tx` with the concrete refinement associated with `impl_index_vec_i32`, and hence, checks the precondition on the index `i`, namely that `i < vec.len`.

3 A Core Calculus of Generic Refinements

Next, we formalize the key aspects of generic refinement types in a core calculus λ_G, an extension of the simply typed λ-calculus with generic refinements.

3.1 Syntax

Fig. 7 summarizes the syntax of λ_G. We stratify the syntax into two layers: refinements (and their sorts), and expressions (and their types). Our approach is generic over an underlying SMT decidable logic used for refinements. Since our system is syntactic in nature, we fix a logic of linear arithmetic and uninterpreted functions for illustration.

Sorts A *base sort* σ_b is either an integer ($\mathbb{Z}$) or a boolean ($\mathbb{B}$). A *sort* σ generalizes base sorts to also include function sorts. We distinguish between base and function sorts as, to preserve SMT-based decidable checking, our algorithmic system will require that generic refinements are only instantiated with refinement-level functions that take in base-sorted arguments (§ 3.2).

Refinements The simplest refinement is an *atom* which is either an integer literal z, the boolean literals tt (true) or ff (false), or a *refinement variable* a. Further, we can combine refinements using the boolean connectives $\wedge$, $\vee$, and $\neg$, or the arithmetic operators =, +, −, and >. Finally, λ_G also includes *refinement applications* $r\ r$ and *refinement abstractions* $\lambda a : \sigma.\, r$. For example, the `ensures` clause in the signature for `get` in Fig. 2 uses a refinement abstraction $\lambda i : \mathbb{Z}.\, \mathit{available}\ i \vee \neg(i = \mathit{idx})$, where *available* and *idx* are refinement variables.

Types λ_G is parameterized by a family of base types b. For illustration, we assume we have `int` and `bool` as base types. A refinement type $\{b[r_1] \mid r_2\}$ instantiates a base type b with a refinement r_1 attaching a constraint r_2. Unlike a traditional refinement type system, this syntax does not bind a variable. Refinement variables are introduced via *refinement generalization* or *existential types* (discussed next). When writing $\{b[r_1] \mid r_2\}$, the intuitive meaning of r_2 is a predicate that must be true. Meanwhile, λ_G does not have an opinion on the meaning of the refinement application $b[r_1]$. We only assume a base type b has a corresponding sort b_σ and must be applied to refinements of that sort. *Syntactically*, the meaning of a generic refinement is given by its *API* as we discuss later

in this section. In § 5, we give a *semantic* interpretation of generic refinements via a translation into the polymorphic contract calculus [24]. λ_G also has function types $\tau \to \tau$, and a limited form of existential types $\{a.\ b[a] \mid r\}$, denoting a refinement type $\{b[a] \mid r\}$ where a has been existentially quantified. Finally, we write $b[r]$ as short for $\{b[r] \mid \mathtt{tt}\}$ and b as short for $\{a.\ b[a] \mid \mathtt{tt}\}$.

Schemes A *scheme* η is a type τ over which various refinement variables a have been quantified (*i.e.,* bound). We use two *inference modes*—**hrn** and **hdl**—to respectively denote whether a given refinement variable a is a **Horn** generic refinement (§ 2.1) or a **Hindl**ey generic refinement (§ 2.2). Note that unlike many formulations of refinement types, we use generic refinements and quantification instead of dependent function binders. For example, in a system like the one in [14], we can write a dependent function type $(x : \mathtt{int}) \to \{v : \mathtt{int} \mid v < x\}$ denoting a function that takes an integer x and returns some other integer less than x. In λ_G, the same function is specified via the scheme that generalizes the input integer with a refinement variable a_1 that is then used to constrain the output integer:

$$\forall a_1 :_{\mathbf{hdl}} \mathbb{Z}.\ \mathtt{int}[a_1] \to \{a_2.\ \mathtt{int}[a_2] \mid a_2 < a_1\}$$

Expressions Finally, the expressions e of λ_G include variables x, base constants c (including numbers, booleans, *etc.*), application of primitive operations $\mathrm{op}(e_1, \ldots, e_n)$, lambda abstractions $\lambda x.\ e$, applications $e(e_1, \ldots, e_n)$, and type ascriptions $e : \tau$. Each constant c has a corresponding type $\mathrm{ty}(c)$. Similarly, each operation op has a scheme given by $\mathrm{scheme}(\mathrm{op})$. The set of constants and operations constitute the API defining the meaning of generic refinements. For example, given the base type `int` and its corresponding sort $\mathtt{int}_\sigma = \mathbb{Z}$, we define $\mathtt{int}[r]$ by defining the type of numeric constants and arithmetic operations as follows:

$$\begin{array}{lcl}
\mathrm{ty}(0) & := & \mathtt{int}[0] \\
\mathrm{ty}(42) & := & \mathtt{int}[42] \\
\mathrm{scheme}(+) & := & \forall a_1 :_{\mathbf{hdl}} \mathbb{Z}, a_2 :_{\mathbf{hdl}} \mathbb{Z}.\ \mathtt{int}[a_1] \to \mathtt{int}[a_2] \to \mathtt{int}[a_1 + a_2] \\
\mathrm{scheme}(/) & := & \forall a_1 :_{\mathbf{hdl}} \mathbb{Z}, a_2 :_{\mathbf{hdl}} \mathbb{Z}.\ \mathtt{int}[a_1] \to \{\mathtt{int}[a_2] \mid a_2 \neq 0\} \to \mathtt{int}[a_1/a_2]
\end{array}$$

Similarly, we can define the `Query` type discussed in § 2.2 assuming it is indexed by a *row* sort and giving schemes for the operations in its API, e.g.:

$$\begin{array}{lcl}
\mathrm{scheme}(\mathit{and}) & := & \forall a_1 :_{\mathbf{hdl}} \mathit{row} \to \mathbb{B}, a_2 :_{\mathbf{hdl}} \mathit{row} \to \mathbb{B}. \\
& & \quad \mathtt{Query}[a_1] \to \mathtt{Query}[a_2] \to \mathtt{Query}[\lambda a_3 : \mathit{row}.\ a_1\ a_3 \wedge a_2\ a_3]
\end{array}$$

3.2 Well-formedness

λ_G has refinement-level λ-abstractions and applications, but the system carefully controls where they occur to ensure that type checking only generates decidable, first-order SMT validity queries. We achieve this by ensuring that types are *well-formed*, meaning, intuitively, that (**I1**) Horn generic variables, *i.e.,* bound as $\forall a :_{\mathbf{hrn}} \sigma.\ \eta$, only appear in η under top-level conjunctions where they can be instantiated via constrained Horn clauses at application sites; (**I2**) refinement variables a are either base-sorted, or are functions that take base-sorted inputs; (**I3**) refinement abstractions do not appear arbitrarily nested but are only used as generic arguments where they can be eliminated as part of constraint generation; and (**I4**) Hindley generic variables, *i.e.,* bound as $\forall a :_{\mathbf{hdl}} \sigma.\ \eta$, appear in η in a value-dependent position that guarantees they can be syntactically instantiated via unification at application sites. These invariants are enforced by the well-sortedness and well-formedness judgments whose syntax-directed rules are summarized in Fig. 8 and Fig. 9.

Refinement Contexts A *refinement context* Θ is a sequence of refinement bindings $a :_\mu \sigma$ and refinement constraints r (used during type-checking). We write $\Theta(a) = (\mu, \sigma)$ to extract the mode

Well-sorted Refinements $\boxed{\Theta \vdash_\ell r : \sigma}$

$$\textsc{WS-hrn}\ \frac{\Theta \vdash_\bot r : \sigma_b \quad \Theta(a) = (\mathbf{hrn}, \sigma_b \to \mathbb{B})}{\Theta \vdash_\top a\ r : \mathbb{B}} \qquad \textsc{WS-hdl}\ \frac{\Theta(a) = (\mathbf{hdl}, \sigma)}{\Theta \vdash_\ell a : \sigma} \qquad \textsc{WS-Int}\ \frac{}{\Theta \vdash_\ell z : \mathbb{Z}} \qquad \textsc{WS-tt}\ \frac{}{\Theta \vdash_\ell \mathrm{tt} : \mathbb{B}}$$

$$\textsc{WS-ff}\ \frac{}{\Theta \vdash_\ell \mathrm{ff} : \mathbb{B}} \qquad \textsc{WS-And}\ \frac{\Theta \vdash_\ell r_1 : \mathbb{B} \quad \Theta \vdash_\ell r_2 : \mathbb{B}}{\Theta \vdash_\ell r_1 \wedge r_2 : \mathbb{B}} \qquad \textsc{WS-Or}\ \frac{\Theta \vdash_\bot r_1 : \mathbb{B} \quad \Theta \vdash_\bot r_2 : \mathbb{B}}{\Theta \vdash_\ell r_1 \vee r_2 : \mathbb{B}} \qquad \textsc{WS-Not}\ \frac{\Theta \vdash_\bot r : \mathbb{B}}{\Theta \vdash_\ell \neg r : \mathbb{B}}$$

$$\textsc{WS-Rel}\ \frac{\Theta \vdash_\bot r_1 : \sigma_b \quad \Theta \vdash_\bot r_2 : \sigma_b}{\Theta \vdash_\ell r_1 \bowtie r_2 : \mathbb{B}} \qquad \textsc{WS-Op}\ \frac{\Theta \vdash_\bot r_1 : \mathbb{Z} \quad \Theta \vdash_\bot r_2 : \mathbb{Z}}{\Theta \vdash_\ell r_1 \oplus r_2 : \mathbb{Z}} \qquad \textsc{WS-App}\ \frac{\Theta \vdash_\bot r_1 : \sigma_b \to \sigma'_b \quad \Theta \vdash_\bot r_2 : \sigma_b}{\Theta \vdash_\ell r_1\ r_2 : \sigma'_b}$$

Fig. 8. Well-Sorted Refinements

μ and sort σ associated to a variable. When tracking variables introduced by λ-abstractions we omit the mode from the binding as it is not relevant. Formaly, we assume it has mode **hdl**.

Well-sorted Refinements The well-sortedness judgment $\Theta \vdash_\ell r : \sigma$ says that a refinement r has sort σ at *level* ℓ in a refinement context Θ. The judgment is parameterized by a level ℓ which is one of $\top$ (at the top-level, not under a disjunction, or negation), or $\bot$ (any other position). Fig. 8 shows the rules that establish the well-sortedness judgment. The majority of the rules use the syntax of boolean and arithmetic operators to ensure that the operands are well-sorted, until they hit the base cases for $\mathbb{B}$ or $\mathbb{Z}$ literals, or a variable. WS-**hrn** restricts variables bound with **hrn** mode to be applied to a refinement at the top-level **(I1)**. On the other hand, a refinement variable with **hdl** mode is allowed at any level (WS-**hdl**). WS-app allows refinement abstractions to be well-sorted as long as the input parameter is of base sort (**I2**). Note that lambda abstractions are not well-sorted under these rules, but can appear in a generic application as discussed next (**I3**).

Well-formed Generic Application The judgment $\Theta \vdash_p b[r]\ [\Xi]$ (ignore p and Ξ for now) states that the generic application $b[r]$ is well-formed. This essentially means that r is well-sorted with sort b_σ (WF-other). Additionally, we allow first order refinement abstractions to be used as generic arguments (WF-lam). Finally, we further require variables to be used at the appropriate mode (WF-var- and WF-var+). Each base type b is associated with a mode b_μ and it can only be instantiated with variables bound with that mode.

Well-formed Types The *type* well-formedness judgment $\Theta \vdash_p \tau$ type $[\Xi]$ says that the type τ is well-formed under Θ with variables Ξ used in value-dependent position, when occurring under polarity p (either + or −). We call *value-dependent* a position that is tied to a value known at call-sites. Variables used in these positions are called value-dependent. A value-dependent variable is guaranteed to be unifiable when calling a function (**I4**). Our value-dependent notion is related to the value-determined notion of Economou et al. [10], however, they are concerned with indexes being *uniquely* determined by their use, in a semantic sense, whereas we take a more syntactic view where our primary focus is in automatically instantiating refinement generics at call-sites.

Fig. 9 summarizes the rules establishing the well-formedness judgment. WF-reft checks that a refinement type $\{b[r_1] \mid r_2\}$ is well-formed by checking the application $b[r_1]$ is well-formed and the constraint r_2 is bool-sorted. In the case where r_1 is a refinement variable, the generic application judgment will add the variable to the set of value-dependent variables if the type occurs negatively

Well-formed Generic Application $\boxed{\Theta \vdash_{\pm} b[r]\ [\Xi]}$

$$\text{WF-VAR-}\ \frac{\Theta(a) = (b_\mu, b_\sigma)}{\Theta \vdash_{-} b[a]\ [a]} \qquad \text{WF-VAR+}\ \frac{\Theta(a) = (b_\mu, b_\sigma)}{\Theta \vdash_{+} b[a]\ [\cdot]}$$

$$\text{WF-LAM}\ \frac{b_\sigma = \sigma_b \to \sigma'_b \qquad \Theta, a : \sigma_b \vdash_{\top} r : \sigma'_b}{\Theta \vdash_{\pm} b[\lambda a : \sigma_b.\, r]\ [\cdot]} \qquad \text{WF-OTHER}\ \frac{r \neq a \wedge r \neq \lambda a : \sigma.\, r' \qquad \Theta \vdash_{\top} r : b_\sigma}{\Theta \vdash_{\pm} b[r]\ [\cdot]}$$

Well-formed Types $\boxed{\Theta \vdash_{\pm} \tau\ \text{type}\ [\Xi]}$

$$\text{WF-REFT}\ \frac{\Theta \vdash_{\pm} b[r_1]\ [\Xi] \qquad \Theta \vdash_{\top} r_2 : \mathbb{B}}{\Theta \vdash_{\pm} \{b[r_1] \mid r_2\}\ \text{type}\ [\Xi]} \qquad \text{WF-EXISTS}\ \frac{\Theta, a :_{\mathbf{hdl}} b_\sigma \vdash_{\top} r : \mathbb{B}}{\Theta \vdash_{\pm} \{a.\, b[a] \mid r\}\ \text{type}\ [\cdot]}$$

$$\text{WF-FUN}\ \frac{\Theta \vdash_{\neg p} \tau_1\ \text{type}\ [\Xi_1] \qquad \Theta \vdash_{p} \tau_2\ \text{type}\ [\Xi_2]}{\Theta \vdash_{p} \tau_1 \to \tau_2\ \text{type}\ [\Xi_1 \cup \Xi_2]}$$

Well-formed Schemes $\boxed{\Theta \vdash \eta\ \text{sch}\ [\Xi]}$

$$\text{WF-TY}\ \frac{\Theta \vdash_{+} \tau\ \text{type}\ [\Xi]}{\Theta \vdash \tau\ \text{sch}\ [\Xi]} \qquad \text{WF-}\mathbf{HRN}\ \frac{\Theta, a :_{\mathbf{hrn}} \sigma \vdash \eta\ \text{sch}\ [\Xi]}{\Theta \vdash \forall a :_{\mathbf{hrn}} \sigma.\, \eta\ \text{sch}\ [\Xi]} \qquad \text{WF-}\mathbf{HDL}\ \frac{\Theta, a :_{\mathbf{hdl}} \sigma \vdash \eta\ \text{sch}\ [\Xi] \qquad a \in \Xi}{\Theta \vdash \forall a :_{\mathbf{hdl}} \sigma.\, \eta\ \text{sch}\ [\Xi]}$$

Fig. 9. Well-Formed Types and Schemes

(WF-VAR-). For existential types $\{a.\, b[a] \mid r\}$, the rule WF-EXISTS checks that r is bool-sorted in the refinement context Θ extended with a binding for a. Finally, for function types $\tau_1 \to \tau_2$, the rule WF-FUN recursively checks the input and output types are well-formed after flipping the polarity for the input type, and states the set of value-dependent variables is the union of the sets returned for the input and output types.

Well-formed Schemes The *scheme* well-formedness judgment $\Theta \vdash \eta\ \text{sch}\ [\Xi]$ says that the scheme η is well-formed under Θ with value-dependent variables Ξ. Fig. 9 summarizes the rules establishing the judgment. In the base case, the rule WF-TY checks that the scheme τ is well-formed in a positive polarity. In the case where the scheme quantifies over a constraint-generic refinement variable $\forall a :_{\mathbf{hrn}} \sigma.\, \eta$, the rule WF-**HRN** recursively checks the scheme η after suitably extending the refinement context with a. Rule WF-**HDL** is similar, but additionally checks that variables of **hdl** mode are in the set of value-dependent variables.

Example 1: Ill-formed Horn Generics With respect to Horn generics, the main purpose of well-formedness is to ensure they do not appear under a negation or disjunction. For example, consider the following (incorrect) scheme for the *or* operator.

$$\forall a_1 :_{\mathbf{hrn}} \mathit{row} \to \mathbb{B}, a_2 :_{\mathbf{hrn}} \mathit{row} \to \mathbb{B}.\, \texttt{Query}[a_1] \to \texttt{Query}[a_2] \to \texttt{Query}[\lambda a_3 : \mathit{row}.\, a_1\ a_3 \vee a_2\ a_3]$$

This definition is ill-formed because the application $a_1\ a_3$ (resp. $a_2\ a_3$) appears under a disjunction. Concretely, when checking the body of the lambda the level starts at $\top$ (WF-LAM) and then switches to $\bot$ after going through the disjunction (WS-OR). Consequently, the application of the Horn

$$
\begin{array}{rrcl}
\textbf{\textit{Refinements}} & r & ::= & \cdots \mid \hat{a} \mid \kappa(r_1, \ldots, r_n) \\
\textbf{\textit{SMT Terms}} & t & ::= & z \mid \mathtt{tt} \mid \mathtt{ff} \mid a \mid (t_1, \ldots, t_n) \mid \pi_i\ t \\
 & & \mid & t \wedge t \mid t \vee t \mid \neg t \mid t \bowtie t \mid t \oplus t \mid t\ t \\
\textbf{\textit{Predicates}} & p & := & t \mid \kappa(t_1, \ldots, t_n) \mid p \wedge p \\
\textbf{\textit{Constraints}} & \Psi & := & p \mid \Psi \wedge \Psi \mid \forall a : \sigma.\ \Psi \mid \Psi \Rightarrow \Psi \\
\textbf{\textit{Evar context}} & \Delta & ::= & \cdot \mid \Delta, \hat{a} : \sigma \mid \Delta, \hat{a} : \sigma = r
\end{array}
$$

Fig. 10. Syntax of Constraints and Environments for Algorithmic Typing

parameter is not accepted because WS-**HRN** only accepts applications of Horn parameters when the level is $\top$.

Example 2: Ill-formed Hindley Generics The definition above would be well-formed with Hindley generics because they are allowed under a disjunction. However, Hindley generics must appear in a value-dependent position which imposes a different trade off. Intuitively, this means they are used in a generic application in input position. For instance, consider a function that computes the maximum between two integers. We may wish to abstract over some predicate p satisfied by both inputs and specify that the output also satisfies it. We could try to define this by abstracting over p with a Hindley generic:

$$\forall p :_{\mathbf{hdl}} \mathbb{Z} \to \mathbb{B}.\ \{a.\ \mathtt{int}[a] \mid p\ a\} \to \{a.\ \mathtt{int}[a] \mid p\ a\} \to \{a.\ \mathtt{int}[a] \mid p\ a\}$$

This definition is ill-formed because p is not used in a generic application in any of the inputs. Concretely, the parameter p does not appear in the set Ξ of value-dependent variables because WF-VAR- never applies.

4 Algorithmic Typing

The key challenge that generic refinements pose is that of instantiating the generic refinement parameters at call-sites. (Requiring the programmer spell those out would make the system rather too tiresome to use.) Next, we present an algorithmic type checker for λ_G that automatically synthesizes the instantiations of generic refinements.

All the rules in the algorithmic system presuppose well-formedness of their inputs. This ensures that generic refinements, both Hindley and Horn, can be automatically instantiated at function calls. Note that well-formedness is not required for soundness (§ 5.3) but to guarantee predictable inference. We could have instead chosen to fail "lazily" at call-sites if generics refinements cannot be instantiated.

4.1 Inference Variables and Constraints

We extend the syntax of refinements with two types of *inference variables* (Fig. 10) used to instantiate parameters in the two possible modes (**hdl** and **hrn**).

Instantiating Hindley Parameters The first kind of variables are *evars* $\hat{a}$: *existential* refinement variables [10] that stand for the unknown indices that a **hdl**-generic refinement parameter can be instantiated with at a call-site, not dissimilar to existential variables used in Rocq [4]. Specifically, the algorithmic typing rules introduce evars at call-sites, and use *Evar Contexts* Δ (Fig. 10) to track their sorts ($\hat{a} : \sigma$) and optionally, the refinement r they have been solved to ($\hat{a} : \sigma = r$).

Instantiating Horn Parameters The second kind of variables are *Horn* variables κ that represent unknown relations (predicates) over different refinements. Horn variables are used at call-sites to instantiate **hrn**-generic refinement parameters, which cannot be directly solved via unification

Subtyping $\boxed{\Theta; \Delta_1 \vdash \tau_1 <: \tau_2 \dashv \Delta_2; \Psi}$

$$\frac{\Theta; \Delta \vdash r_1 \equiv [\Delta] r_2 : b_\sigma \dashv \Delta'; \Psi}{\Theta; \Delta \vdash \{b[r_1] \mid \mathrm{tt}\} <: \{b[r_2] \mid \mathrm{tt}\} \dashv \Delta'; \Psi} \;<:\textsc{eq}$$

$$\frac{\Theta; \Delta \vdash \tau_1' <: \tau_1 \dashv \Delta'; \Psi_1 \qquad \Theta; \Delta' \vdash \tau_2 <: \tau_2' \dashv \Delta''; \Psi_2}{\Theta; \Delta \vdash \tau_1 \to \tau_2 <: \tau_1' \to \tau_2' \dashv \Delta''; \Psi_1 \wedge \Psi_2} \;<:\textsc{fun}$$

$$\frac{\Theta, r_2; \Delta \vdash \{b[r_1] \mid \mathrm{tt}\} <: \tau \dashv \Delta'; \Psi}{\Theta; \Delta \vdash \{b[r_1] \mid r_2\} <: \tau \dashv \Delta'; r_2 \Rightarrow \Psi} \;<:\textsc{reft/l}$$

$$\frac{\Theta; \Delta \vdash \tau <: \{b[r_1] \mid \mathrm{tt}\} \dashv \Delta'; \Psi}{\Theta; \Delta \vdash \tau <: \{b[r_1] \mid r_2\} \dashv \Delta'; [\Delta'] r_2 \wedge \Psi} \;<:\textsc{reft/r}$$

$$\frac{\Theta, a :_{\mathbf{hdl}} b_\sigma; \Delta \vdash \{b[a] \mid r\} <: \tau \dashv \Delta'; \Psi}{\Theta; \Delta \vdash \{a.\, b[a] \mid r\} <: \tau \dashv \Delta'; \forall a : b_\sigma.\, \Psi} \;<:\textsc{exists/l}$$

$$\frac{\Theta; \Delta, \hat{a} : b_\sigma \vdash \tau <: \{b[\hat{a}] \mid r[\hat{a}/a]\} \dashv \Delta, \hat{a} : b_\sigma = r'; \Psi}{\Theta; \Delta \vdash \tau <: \{a.\, b[a] \mid r\} \dashv \Delta; \Psi} \;<:\textsc{exists/r}$$

Refinement Equivalence $\boxed{\Theta; \Delta \vdash r_1 \equiv r_2 : \sigma \dashv \Delta'; \Psi}$

$$\frac{r \text{ ground} \quad \Delta \triangleq \Delta_1, \hat{a} : \sigma', \Delta_2}{\Theta; \Delta \vdash r \equiv \hat{a} : \sigma \dashv \Delta_1, \hat{a} : \sigma' = r, \Delta_2; \mathrm{tt}} \equiv\textsc{inst}$$

$$\frac{\forall \hat{a} \in dom(\Delta).\ r_2 \neq \hat{a}}{\Theta; \Delta \vdash r_1 \equiv r_2 : \sigma_b \dashv \Delta; r_1 = r_2} \equiv\textsc{base}$$

$$\frac{\Psi \triangleq \kappa(\overline{r}) \Rightarrow r' \wedge r' \Rightarrow \kappa(\overline{r})}{\Theta; \Delta \vdash \kappa(\overline{r}) \equiv r' : \mathbb{B} \dashv \Delta; \Psi} \equiv\kappa/\textsc{l} \qquad \frac{\Psi \triangleq \kappa(\overline{r}) \Rightarrow r' \wedge r' \Rightarrow \kappa(\overline{r})}{\Theta; \Delta \vdash r' \equiv \kappa(\overline{r}) : \mathbb{B} \dashv \Delta; \Psi} \equiv\kappa/\textsc{r}$$

$$\frac{\Theta, a : \sigma_b; \Delta \vdash r_1\ a \equiv r_2\ a : \sigma_b' \dashv \Delta'; \Psi}{\Theta; \Delta \vdash r_1 \equiv r_2 : \sigma_b \to \sigma_b' \dashv \Delta; \forall a : \sigma_b.\, \Psi} \equiv\textsc{fun}$$

Fig. 11. Subtyping and Refinement Equivalence

as they require a fixpoint computation. Our algorithmic typing rules instantiate **hrn** parameter by following the recipe of liquid typing [6, 23] introducing a fresh Horn variable to represent the unknown predicate. Using applications $\kappa(r_1, \ldots, r_n)$ of unknown predicates, the system builds a Horn Constraint Ψ whose solution yields a valid instantiation of the generic refinement parameters. Note that constraints Ψ are built from a restricted set of SMT terms t that do not contain refinement λ-abstractions. The type checking rules guarantee that abstractions are eliminated from refinements when generating the constraint. Moreover, note that invariant (**I1**), ensured by well-formedness (§ 3.2), guarantees that we get a valid Horn constraint by verifying that applications of **hrn** parameters only appear at the top-level, and *not* under negations or disjunctions.

4.2 Subtyping

λ_G uses an algorithmic subtyping judgment of the form $\Theta; \Delta_1 \vdash \tau_1 <: \tau_2 \dashv \Delta_2; \Psi$ which says that the type τ_1 is a *subtype of* τ_2, in the refinement context Θ using the evar context Δ_1, with the evar context updated to Δ_2, *if* the Horn constraint Ψ is satisfiable.

Subtyping Rules Fig. 11 shows the syntax-directed rules that establish the subtyping judgment. The rule <:FUN is the usual one for function types that recurses contra-variantly on inputs and co-variantly on outputs, conjoining the constraints Ψ_1 and Ψ_2 that are generated by each. In rule <:REFT/L, the predicate r_2 appears on the left, and it is then added as an *assumption* that can be used to prove the rest of the constraint Ψ. Dually, in rule <:REFT/R, r_2 appears on the right and has

to be *proven*. Thus, it is conjoined with the constraint Ψ. Note that we substitute in r_2 all solved evars in Δ' with their solution (written $[\Delta']r_2$). This is to ensure that evars are eagerly resolved, which we maintain as an invariant of the system. Rule <:EXISTS/L adds the existential variable to the context by universally quantifying over it in the resulting constraint. The dual rule, <:EXISTS/R, must find a suitable refinement to instantiate the refinement variable on the right. To this end, the variable is replaced by an evar whose solution must be found by the recursive invocation of the rule as specified by the output evar context in the premise. Finally, the rule <:EQ delegates the work of checking generic arguments to the refinement equivalence judgment.

Refinement Equivalence The judgment $\Theta; \Delta \vdash \ r_1 \equiv r_2 : \sigma \ \dashv \Delta; \Psi$ checks if two refinements are equivalent at sort σ. For function sorts (≡FUN), we verify that the equivalence holds extensionally by quantifying over the input variable and ensuring that the refinements applied to this variable are equivalent at the output sort. To check for equivalence at bool when Horn variables are involved, we generate a double implication (<: κ/L and <: κ/R) ensuring the constraint is a well-formed Horn constraint. The rule ≡INST *instantiates* an evar $\hat{a}$ that is still not solved in Δ by unifying it with a *ground* refinement r that does not contain any evars. Finally, ≡BASE kicks in when r_2 is not an evar, in which case it generates a constraint requiring r_1 and r_2 to be equal.

4.3 Typing

Finally, λ_G uses a bidirectional algorithmic typing judgment inspired by the system of Economou et al. [10]. The rules establishing the judgments are shown in Fig. 12. The type checking judgment $\Theta; \Gamma; \Delta \vdash \ e \ \Leftarrow \eta \dashv \Delta'; \Psi$ says that under refinement context Θ, program context Γ (Fig. 7), and evar context Δ, the expression e has the scheme η outputing context Δ' *if* the constraint Ψ is satisfiable.

T-ABS checks a lambda abstraction $\lambda x.\ e$ against a function type $\tau_1 \to \tau_2$ by checking that the body e has type τ_2 in the program context Γ extended with a binding $x : \tau_1$. The subsumption rule T-SUB is mostly standard. We only draw attention to the fact that the output constraints of the synthesis and subtyping judgments are conjoined in the generated constraint. The last rule, T-FORALL, checks the expression e against a generic scheme by extending the refinement context and universally quantifying over the refinement variable in the generated constraint.

The synthesis judgment $\Theta; \Gamma \vdash \ e \ \Rightarrow \eta \dashv \Psi$ says that under refinement context Θ and program context Γ the expression e synthesizes scheme η *if* the constraint Ψ is satisfiable. T-VAR says that variables x get their types from the program context Γ, and similarly T-CON says that base constants c are assigned their builtin types $\mathrm{ty}(c)$, and this happens unconditionally, *i.e.,* with the trivial Horn constraint tt. As usual, the ascription rule T-ASC allows switching to check mode.

The interesting action happens in the rules that check applications for functions and primitive operations, as that is where we have to instantiate (*i.e.,* synthesize) the generic refinements for the callee using evars or Horn variables. T-APP checks an application $e(e'_1, \ldots, e'_n)$ by first synthesizing a scheme η for the callee e and then invoking the *function application* judgment to check that when η is *applied to* the arguments $e'_1, \ldots, e'_n$ in an *empty* evar context, the result is the type τ and another empty evar context, which will, informally speaking, ensure that all the evars in η are instantiated to ground refinements. The Horn constraint for the application is then the conjunction of the constraints Ψ_1—generated by the callee's typing—and Ψ_2—generated by the application judgment. The rule T-OP is similar, but uses the builtin scheme for op as the callee's scheme.

Function Applications Fig. 12 shows the rules for establishing the function application judgment $\Theta; \Gamma; \Delta \vdash \ [\ \eta\ ](\overline{e}) \ \gg \tau \dashv \Delta'; \Psi$ which says that when the scheme η is applied to the arguments $\overline{e}$ in the evar context Δ, the result is the type τ, with the evar context updated to Δ', *if* the Horn constraint Ψ holds. (Recall that the evar contexts are used to track which evars need to be instantiated, denoted by $\hat{a} : \sigma$, and which have already been solved, denoted by $\hat{a} : \sigma = r$.) We establish the application

Type Checking $\boxed{\Theta; \Gamma; \Delta \vdash e \Leftarrow \eta \dashv \Delta; \Psi}$

$$\textsc{T-Forall}\ \frac{\Theta, a :_{\mu} \sigma; \Gamma; \Delta \vdash e \Leftarrow \eta \dashv \Delta'; \Psi}{\Theta; \Gamma; \Delta \vdash e \Leftarrow \forall a :_{\mu} \sigma.\, \eta \dashv \Delta'; \forall a : \sigma.\, \Psi} \qquad \textsc{T-Sub}\ \frac{\Theta; \Gamma \vdash e \Rightarrow \tau_1 \dashv \Psi_1 \qquad \Theta; \Delta \vdash \tau_1 <: \tau_2 \dashv \Delta'; \Psi_2}{\Theta; \Gamma; \Delta \vdash e \Leftarrow \tau_2 \dashv \Delta'; \Psi_1 \wedge \Psi_2}$$

$$\textsc{T-Abs}\ \frac{\Theta; \Gamma, x : \tau_1; \Delta \vdash e \Leftarrow \tau_2 \dashv \Delta'; \Psi}{\Theta; \Gamma; \Delta \vdash \lambda x.\, e \Leftarrow \tau_1 \to \tau_2 \dashv \Delta'; \Psi}$$

Type Synthesis $\boxed{\Theta; \Gamma \vdash e \Rightarrow \eta \dashv \Psi}$

$$\textsc{T-Var}\ \frac{\Gamma(x) = \tau}{\Theta; \Gamma \vdash x \Rightarrow \tau \dashv \mathrm{tt}} \qquad \textsc{T-Con}\ \frac{}{\Theta; \Gamma \vdash c \Rightarrow \mathrm{ty}(c) \dashv \mathrm{tt}} \qquad \textsc{T-Asc}\ \frac{\Theta; \Gamma; \cdot \vdash e \Leftarrow \eta \dashv \cdot; \Psi}{\Theta; \Gamma \vdash e : \eta \Rightarrow \eta \dashv \Psi}$$

$$\textsc{T-App}\ \frac{\Theta; \Gamma \vdash e \Rightarrow \eta \dashv \Psi_1 \quad \Theta; \Gamma; \cdot \vdash [\ \eta\ ](e'_1, \ldots, e'_n) \gg \tau \dashv \cdot; \Psi_2}{\Theta; \Gamma \vdash e(e'_1, \ldots, e'_n) \Rightarrow \tau \dashv \Psi_1 \wedge \Psi_2}$$

$$\textsc{T-Op}\ \frac{\mathrm{scheme}(\mathrm{op}) = \eta \quad \Theta; \Gamma; \cdot \vdash [\ \eta\ ](e_1, \ldots, e_n) \gg \tau \dashv \cdot; \Psi}{\Theta; \Gamma \vdash \mathrm{op}(e_1, \ldots, e_n) \Rightarrow \tau \dashv \Psi}$$

Typing Applications $\boxed{\Theta; \Gamma; \Delta \vdash [\ \eta\ ](\overline{e}) \gg \tau \dashv \Delta'; \Psi}$

$$\textsc{FA-hdl}\ \frac{\Theta; \Gamma; \Delta, \hat{a} : \sigma \vdash [\ \eta[\hat{a}/a]\ ](\overline{e}) \gg \tau \dashv \Delta', \hat{a} : \sigma = r; \Psi}{\Theta; \Gamma; \Delta \vdash [\ \forall a :_{\mathbf{hdl}} \sigma.\, \eta\ ](\overline{e}) \gg \tau \dashv \Delta'; \Psi} \qquad \textsc{FA-Res}\ \frac{}{\Theta; \Gamma; \Delta \vdash [\ \tau\ ](\cdot) \gg \tau \dashv \Delta; \mathrm{tt}}$$

$$\textsc{FA-hrn}\ \frac{\kappa \text{ fresh} \quad \mathrm{vars}(\Theta) = \overline{a'} \quad r = \lambda a : \sigma.\, \kappa(a, \overline{a'}) \qquad \Theta; \Gamma; \Delta \vdash [\ \eta[r/a]\ ](\overline{e}) \gg \tau \dashv \Delta'; \Psi}{\Theta; \Gamma; \Delta \vdash [\ \forall a :_{\mathbf{hrn}} \sigma.\, \eta\ ](\overline{e}) \gg \tau \dashv \Delta'; \Psi}$$

$$\textsc{FA-Fun}\ \frac{\Theta; \Gamma; \Delta \vdash e \Leftarrow \tau_1 \dashv \Delta'; \Psi_1 \qquad \Theta; \Gamma; \Delta' \vdash [\ [\Delta']\tau_2\ ](\overline{e'}) \gg \tau' \dashv \Delta''; \Psi_2}{\Theta; \Gamma; \Delta \vdash [\ \tau_1 \to \tau_2\ ](e, \overline{e'}) \gg \tau' \dashv \Delta''; [\Delta'']\Psi_1 \wedge \Psi_2}$$

Fig. 12. Algorithmic Typing for Expressions and Applications

judgment using four rules directed by the syntax of η. FA-**hdl** accounts for Hindley generics $a :_{\mathbf{hdl}} \sigma$ by substituting a with a fresh evar $\hat{a}$ that is added (unsolved) to the evar context in which the (substituted) scheme is checked. Crucially, the rule also requires that in the resulting evar context, $\hat{a}$ is solved to some refinement r. FA-**hrn** handles Horn generics $a :_{\mathbf{hrn}} \sigma$ by creating a fresh Horn variable κ, and replacing a with a Horn application $\lambda a : \sigma.\, \kappa(a, \overline{a'})$ in the scheme η. FA-Fun kicks in after the generics have been instantiated, and checks that the (first) argument e has the expected input type τ_1, by first typing e as τ, then checking τ is a subtype of τ_1, and finally

checking the rest of the arguments $\overline{e'}$ against the output τ_2. Finally, FA-Res applies when we have checked all the arguments and simply says the result is the (residual) scheme τ.

Example To illustrate the rules, let us recall the example in Fig. 5 and see how Hindley generics are instantiated when combining the subqueries with the and method. As described in § 3.1, we model the `Query` type assuming it is indexed by a `row` sort and giving the *and* operation the scheme:

$$\forall a_1 :_{\mathbf{hdl}} row \to \mathbb{B}, a_2 :_{\mathbf{hdl}} row \to \mathbb{B}.\ \texttt{Query}[a_1] \to \texttt{Query}[a_2] \to \texttt{Query}[\lambda a_3 : row.\ a_1\ a_3 \wedge a_2\ a_3]$$

Now, consider the application $and(q_1, q_2)$ where q_1 and q_2 have the types shown in Fig. 5.

$$\begin{array}{rcl} q_1 & : & \texttt{Query}[\lambda r : row.\ item\ r = owner] \\ q_2 & : & \texttt{Query}[\lambda r : row.\ price\ r < cost] \end{array}$$

To check the function application, we begin by using FA-**HDL** twice to eliminate refinement parameters generating two evars, $\hat{a}_1$ and $\hat{a}_2$, for the corresponding parameters a_1 and a_2. This leaves us with the following residual:

$$\texttt{Query}[\hat{a}_1] \to \texttt{Query}[\hat{a}_2] \to \texttt{Query}[\lambda a_3 : row.\ \hat{a}_1\ a_3 \wedge \hat{a}_2\ a_3]$$

Then, we use rule FA-Fun to check the first argument q_1. This requires checking the subtyping $\texttt{Query}[\lambda r : row.\ item\ r = owner] <: \texttt{Query}[\hat{a}_1]$. By unifying the indices (≡inst) we can solve $\hat{a}_1$ to $\lambda r : row.\ item\ r = owner$. A subsequent application of FA-Fun solves $\hat{a}_2$ to $\lambda r : row.\ price\ r < cost$. Substituting the solution to $\hat{a}_1$ and $\hat{a}_2$ in the output type gives us:

$$\texttt{Query}[\lambda a_3 : row.\ (\lambda r : row.\ item\ r = owner)\ a_3 \wedge (\lambda r : row.\ price\ r < cost)\ a_3]$$

Finally, we eliminate the lambdas by applying beta reduction to obtain:

$$\texttt{Query}[\lambda a_3 : row.\ item\ a_3 = owner \wedge price\ a_3 < cost]$$

We note that well-formedness (§ 3.2) guarantees that lambdas can always be eliminated via beta reduction ensuring we only generate first-order SMT validity queries. Moreover, the value-dependent restriction ensures that all evars can be solved after checking the arguments in a function call.

5 Semantics of λ_G

In § 3, we described how we can define the meaning of generic refinements via the set of primitive constants and operations on base types. Next, we give a semantic interpretation of generic refinements by noting that we can think of them as *ghost* variables whose values are automatically synthesized at function calls. We formalize this notion by translating λ_G into the polymorphic contract calculus $\mathrm{F}^{\sigma}_{\mathrm{H}}$ of Sekiyama et al. [24]. For space restrictions, we only give a high-level description of the translation and its properties. The complete definitions and proofs can be found in the accompanying technical appendix [18].

5.1 Refinement Elaboration

To translate λ_G into $\mathrm{F}^{\sigma}_{\mathrm{H}}$ we first define an intermediate calculus $\lambda_{\underline{G}}$ where refinement instantiations are explicit. Expressions $\underline{e}$ in $\lambda_{\underline{G}}$ follow the syntax of λ_G with the difference that applications are instantiated with an explicit list of refinements:

$$\textbf{\textit{Expressions}} \quad \underline{e} \ ::= \ x \mid c \mid \lambda x : \tau.\ \underline{e} \mid \mathrm{op}\ \langle \overline{r} \rangle\ (\overline{\underline{e}}) \mid \underline{e}\ \langle \overline{r} \rangle\ (\overline{\underline{e}}) \mid \underline{e} : \tau$$

Next, we extend the algorithmic checking and sysntehsis judgments into elaboration judgments that insert refinement instantiations at function calls.

$$\Theta; \Gamma; \Delta_1 \vdash e \Leftarrow \eta \dashv \Delta_2; \Psi \rightsquigarrow \underline{e} \qquad \Theta; \Gamma \vdash e \Rightarrow \eta \dashv \Psi \rightsquigarrow \underline{e}$$

Solution of Horn Vars The elaborated expresion $\underline{e}$ may contain unsolved Horn variables. A Horn constraint *solution* Σ is a list of refinements to be substituted for Horn variables: $\Sigma ::= \cdot \mid \Sigma, r/\kappa$. The operation $[\Sigma]\underline{e}$ applies the substitution Σ to an expression $\underline{e}$. Using the algorithm in [6] we can write a procedure $\texttt{Solve} :: \Psi \to \Sigma \cup \bot$ that given a constraint Ψ either returns a solution or determines the constraint to be unsatisfiable. We additionally extend the `Solve` procedure to take a refinement context Θ as follows:

$$\texttt{Solve}(\Theta, a{:}\sigma; \Psi) := \texttt{Solve}(\Theta; \forall a{:}\sigma.\ \Psi) \qquad \texttt{Solve}(\Theta, r; \Psi) := \texttt{Solve}(\Theta; r \Rightarrow \Psi)$$

$$\texttt{Solve}(\cdot; \Psi) := \texttt{Solve}(\Psi)$$

Solution of Hindley Vars To handle evars, we define an *extension* relation for evar contexts $\Theta \vdash \Delta \longrightarrow \Delta'$ that intuitively stipulates that Δ' has more solved variables than Δ. Aditionally, we say that an evar context is *complete*, written Ω, if it only contains solved evars.

Properties of the Algorithmic System The checking $\Theta; \Gamma \vdash \underline{e} \Leftarrow \eta$ and synthesis $\Theta; \Gamma \vdash \underline{e} \Leftarrow \eta$ judgments of $\lambda_{\underline{G}}$ mostly mirror the rules of the algorithmic system but instead of generating a constraint, they rely on a declarative subtyping judgment $\Theta \vdash \tau_1 <: \tau_2$. We show that the algorithmic system is sound with respect to the declarative system.

Theorem 5.1 (Soundness of Algorithmic Typing).

(1) *If* $\Theta; \Gamma \vdash e \Rightarrow \eta \dashv \Psi \rightsquigarrow \underline{e}$ *and* $\texttt{Solve}(\Theta; \Psi \wedge \Psi') = \Sigma$ *then* $[\Sigma]\Theta; [\Sigma]\Gamma \vdash [\Sigma][\Omega]\underline{e} \Rightarrow [\Sigma][\Omega]\eta$
(2) *If* $\Theta; \Gamma; \Delta \vdash e \Leftarrow \eta \dashv \Delta'; \Psi \rightsquigarrow \underline{e}$ *and* $\texttt{Solve}(\Theta; \Psi \wedge \Psi') = \Sigma$ *and* $\Theta \vdash \Delta' \longrightarrow \Omega$ *then* $[\Sigma]\Theta; [\Sigma]\Gamma \vdash [\Sigma][\Omega]\underline{e} \Leftarrow [\Sigma][\Omega]\eta$

The algorithmic system is fundamentally not complete because the Horn constraint solver is not complete [6]. Following Economou et al. [10], we could prove completeness for the solution of Hindley variables by relying on the well-formedness of types, which gurantees Hindley variables can be unified at call-sites. However, we do not consider this aspect fundamental to our approach, and we are considering relaxing the definition of value-dependent to allow parameters that cannot be solved locally.

5.2 Translation Into $\mathrm{F}^{\sigma}_{\mathrm{H}}$

Finally, we define the semantics of generic refinements by translating $\lambda_{\underline{G}}$ into the polymorphic contract calculus $\mathrm{F}^{\sigma}_{\mathrm{H}}$ [24]. $\mathrm{F}^{\sigma}_{\mathrm{H}}$ is parametric over a family of base types B, constants, and primitive operations. We instantiate the system to include sorts as base types and refinements as expressions. Refinement constants (integers and booleans) and operations (arithmetic and boolean connectives) are included as primitive constants and operations in $\mathrm{F}^{\sigma}_{\mathrm{H}}$. Thus, we give meaning to refinements via their operational behavior in $\mathrm{F}^{\sigma}_{\mathrm{H}}$. Additionally, we include base types b as part of base types in $\mathrm{F}^{\sigma}_{\mathrm{H}}$, and constants and operations in λ_G as constants and operations in $\mathrm{F}^{\sigma}_{\mathrm{H}}$.

We translate schemes and types in λ_G into types in $\mathrm{F}^{\sigma}_{\mathrm{H}}$ as follows:

$$\begin{array}{rcl@{\qquad}rcl}
(\!|\forall x : \sigma.\ \eta|\!) & = & (x{:}\,\sigma) \to (\!|\eta|\!) & (\!|\{a.\ b[a] \mid r\}|\!) & = & b \times \{a{:}\,b_\sigma \mid r\} \\
(\!|\{b[r_1] \mid r_2\}|\!) & = & b \times \{x{:}\,(\!|r|\!)_{b_\sigma} \mid r_2\} & (\!|r|\!)_{\sigma_b \to \sigma'_b} & = & (x{:}\,\sigma_b) \to (\!|r\ x|\!)_{\sigma'_b} \\
(\!|\tau_1 \to \tau_2|\!) & = & (\!|\tau_1|\!) \to (\!|\tau_2|\!) & (\!|r|\!)_{\sigma_b} & = & \{x{:}\,\sigma_b \mid x = r\}
\end{array}$$

Most notably, a generic application $b[r]$ is translated into a pair carrying a *ghost* value. The ghost value, which can be of function type, must be (extensionally) equal to r as denoted by $(\!|r|\!)_\sigma$. This translation is naturally extended to refinement contexts $(\!|\Theta|\!)$ and program contexts $(\!|\Gamma|\!)$.

Finally, we translate expressions $\underline{e}$ in $\lambda_{\underline{G}}$ to expressions M in $\mathrm{F}^{\sigma}_{\mathrm{H}}$ via an elaboration judgment $\Theta; \Gamma \vdash \underline{e} : \eta \rightsquigarrow M$ that inserts explicit *casts* as $\mathrm{F}^{\sigma}_{\mathrm{H}}$ does not have subsumption. We show the translation is type preserving.

Theorem 5.2 (Type Preserving Translation).

(1) *If* $\Theta; \Gamma \vdash \underline{e} \Rightarrow \eta \rightsquigarrow M$ *then* $(\!|\Theta|\!), (\!|\Gamma|\!) \vdash_H M : (\!|\eta|\!)$
(2) *If* $\Theta; \Gamma \vdash \underline{e} \Leftarrow \eta \rightsquigarrow M$ *then* $(\!|\Theta|\!), (\!|\Gamma|\!) \vdash_H M : (\!|\eta|\!)$

5.3 Soundness

The dynamic semantics of λ_G is defined by the translation into $\mathrm{F}^{\sigma}_{\mathrm{H}}$. As mentioned in § 5.2, the translation inserts casts whenever subsumption is required in λ_G. Since these casts may fail at runtime, the type-preserving translation alone does not guarantee the soundness of λ_G. However, since the translation only inserts casts between types related by subtyping—verified statically via SMT based implication—these casts should, in principle, never fail.

Following Vazou et al. [28], we can formalize this intuition using the *upcast lemma* from the contract calculus [24]. This lemma states that casts related by subtyping are essentially a no-op (they are logically related to the identity function) and therefore cannot fail. Unfortunately, Sekiyama et al. [24] only conjectured, but did not prove, the upcast lemma. The lemma was proved for an earlier system [3] which had meta-theoretical issues that $\mathrm{F}^{\sigma}_{\mathrm{H}}$ later resolved. Sekiyama et al. [24] speculated that the upcast lemma would hold for $\mathrm{F}^{\sigma}_{\mathrm{H}}$ as well, given that the definition of parametricity remained unchanged, but did not check it because it was not their focus. Assuming the upcast lemma holds, we can combine theorem 5.2 and the type soundness theorem of $\mathrm{F}^{\sigma}_{\mathrm{H}}$ to conjecture soundness for λ_G ensuring well-typed translated programs will never raise an error.

Conjecture 5.3 (Type Soundness). *Suppose* $\cdot; \cdot; \cdot \vdash e \Leftarrow \eta \dashv \cdot; \Psi \rightsquigarrow \underline{e}$ *and* $\mathtt{Solve}(\Psi) = \Sigma$ *and* $\cdot; \cdot \vdash [\Sigma]\underline{e} \Leftarrow [\Sigma]\eta \rightsquigarrow M$. *If* $M \longrightarrow^* M'$ *and* M *does not reduce, then* M' *is a value.*

6 Implementation and Case Studies

We implemented the inference rules from § 4 as an extension to Flux [16]. To evaluate the effectiveness of our approach, we conducted two case studies: one demonstrating the use of generic refinement to verify safe vector access, and the other verifying invariants in database queries

Implementation The core calculus λ_G described in previous sections, describes a simple *model* of generic refinements in a pure setting. Our actual implementation in Flux has to account for all of Rust's imperative features, including references, mutable state, and lifetimes. Fortunately, previous work on Flux already shows how Rust's type system can be used to cleanly separate the imperative features and define refinements on pure values (indices) [16]. As refinement generics operate largely on indices—they can be thought of as function valued indices—the separation from the previous work carries over directly, allowing us to implement generic refinements as an extension to the kinds of indices that were previously handled by Flux.

6.1 Vector Bounds

The Rust standard library provides a vector type, `Vec`, which is widely used across Rust code bases. In [16], the authors describe a way to implement a new `RVec` type that wraps `Vec`, and refine its API to ensure, at compile time, that reads and writes stay within the vector's bounds. However, to use `RVec` the programmer would have to replace all the instances of `Vec` with `RVec`, which is a non-trivial task, and makes that approach a non-starter in practice. Our first case study uses generic associated refinements to refine the `Vec` type, thereby allowing the programmer to use the standard library's vector type while using Flux to get compile-time bounds checking.

Specification We follow the lightweight specification from [16] and index each `Vec` with an integer `len` that represents the vector's length. Vector reads `v[i]` and writes `v[i] = e` are desugared by the Rust compiler roughly into calls `index(v, i)` and `*index_mut(v, i) = e` where `index` and `index_mut`

Vectors	LOC	Spec	Time (s)
std::vec	65	—	—
bsearch	25	0	0.18
dotprod	12	1	0.14
fft	162	7	0.51
heapsort	42	3	0.22
simplex	143	14	0.50
kmeans	85	8	0.37
kmp	48	2	0.35

Queries	LOC	Spec	Time (s)
RDiesel	434	—	—
conf	251	50	0.92
wishlist	176	43	0.68
course	271	49	0.91
voltron	185	25	0.51

Fig. 13. Experimental results using Generic Refinements for checking `std::vec` bounds and RDiesel database queries. **LOC** is the number of lines of Rust *source code*, **Spec** is the number of lines for function *specifications*, **Time (s)** is the time in seconds required to verify the code (trusted code does not have time). The **LOC** reported for `std::vec` and RDiesel are a combination of the intertwined specifications and Rust code.

are methods of the trait `Index` and `IndexMut` respectively that are implemented by the vector type. We use associated refinements to specify the preconditions for these methods, as described in § 2.3.

API Coverage Our benchmarks focus on a subset of the `Vec` API, including `new`, `push`, `pop`, `len`, `index` and `index_mut`. While we have not extensively tested the entire API, many other methods, such as `as_slice`, `insert`, and `split_off`, can be supported without problems. Some operations that involve more advance features (e.g, `pop_if`) are not currently supported but could be added with future extensions to Flux. Finally, for some operations that depend on the specific values stored in the vector, which our lightweight specification does not track, we cannot specify the exact length after the call. For example, we cannot specify the exact length of a vector after a call to `dedup`, but we can say that the length is at least less than on equal.

Verification The left table in Fig. 13 summarizes the results of using associated refinements to verify uses of `Vec` in a suite of loop- and vector-heavy benchmarks from [16] which does bound checking using a bespoke `RVec` type. The LOC denotes non-comment source lines, and SPEC denotes the lines of function contracts. (There are no loop invariants needed as Flux infers them automatically via liquid typing.) The verification is quite efficient, with the most complex benchmark taking well under a second to verify.

6.2 Database Access Control via Refined Diesel

Database-backed web applications use ORM libraries like Rust's Diesel library [26] to dynamically generate queries to access and update sensitive user information. Our second case study is an implementation of RDiesel, a library that uses generic refinements to statically track the semantics of database queries enabling the verification of security properties of web applications in the style of the Storm framework [17]. We present this case study by first sketching the Diesel API § 6.2.1, then showing how we refine it to statically track the semantics of queries § 6.2.2, and finally, how we can use the refinements to specify and verify access control policies § 6.2.3.

6.2.1 The Diesel Query DSL. In § 2.2 we saw a simple DSL where every `Query<R>` evaluated to a `bool` valued result on an `R`-typed row. This simplification meant that atomic comparisons `eq` and `gt` could only be performed on program values, but not *other* fields of the database, and that the DSL disallowed arithmetic on database fields, making it impractical for real-world applications.

Typed Queries via Traits To support such use-cases, Rust's Diesel ORM framework [26] implements a *typed* query DSL that allows for—and statically tracks—different *types* of sub-query

results. Instead of a single `bool` valued `struct Query<R>`, Diesel defines a `trait Expr<R, V>` that represents queries over the row `R` that *evaluate* to a value of type `V`.

```
trait Expr<R, V> { ... }
```

Program values can be "lifted" up into queries by implementing the `Expr` trait for the underlying values, *e.g.,* to treat `i32` and `usize` values as `Expr<R, i32>` and `Expr<R, usize>` Diesel implements

```
impl<R> Expr<R, i32> for i32 { ... }
impl<R> Expr<R, usize> for usize { ... }
```

Representing Rows and Fields As discussed in Section § 2.2, a schema defined by `table!{ ... }` automatically generates a `struct` that represents a row in the underlying table. However, rather than representing fields as values of a single type `Field`, it generates a distinct `struct` for each field. Along with this, it provides a suitable implementation of `Expr`, thereby representing each field at the type level. As discussed in § 2.2, a `table!{ ... }` schema automatically generates a `struct` representing a row in the underlying table. However, instead of representing fields as values of a single type `Field`, it represents them at the type level by generating a distinct `struct` for each one together with suitable implementation of `Expr`.

```
struct ItemId;                impl Expr<Item, usize> for ItemOwner { ... }
struct ItemOwner;             impl Expr<Item, usize> for ItemId    { ... }
struct ItemPrice;             impl Expr<Item, i32>   for ItemPrice { ... }
struct ItemDescr;             impl Expr<Item, String> for ItemDescr { ... }
```

Query Composition The indirection of the `trait Expr` allows Diesel to build a *typed* Query DSL by defining separate `struct`s for different kinds of queries, and suitably constraining the `Expr` implementations for those structs:

```
struct Add<A, B> { lhs: A, rhs: B }
fn add<A, B>(a: A, b: B) -> Add<A, B>;

impl<R, A, B> Expr<R, i32> for Add<A, B>
where A: Expr<R, i32>, B: Expr<R, i32>;

struct Eq<V, A, B> { lhs: A, rhs: B, ... };
fn eq<V, A, B>(a: A, b: B) -> Eq<V, A, B>;

impl<R, V, A, B> Expr<R, bool> for Eq<V, A, B>
where A: Expr<R, V>, B: Expr<R, V>;

struct Gt<V, A, B> { lhs: A, rhs: B, ... }
fn gt<V, A, B>(a: A, b: B) -> Gt<V, A, B>;

impl<R, V, A, B> Expr<R, bool> for Gt<V, A, B>
where A: Expr<R, V>, B: Expr<R, V>;

struct And<A, B> { lhs: A, rhs: B }
fn and<A, B>(a: A, b: B) -> And<A, B>;

impl<R, A, B> Expr<R, bool> for And<A, B>
where A: Expr<R, bool>, B:Expr<R, bool>;
```

Each of `Add`, `Eq`, `Gt` and `And` contain two sub-expressions corresponding to the fields `lhs` and `rhs`. Crucially, the corresponding `impl` blocks ensure that only "well-typed" queries have valid `Expr` implementations. For example, `Add` requires that the two sub-queries are `i32`-valued, producing an `i32` valued `Expr`; `Eq` and `Gt` require the two sub-queries are `V`-valued, producing a `bool` valued `Expr` and `And` requires two `bool` valued sub-queries, producing a `bool` valued result.[2]

6.2.2 Refining the Diesel API. Database-backed applications manipulate sensitive user data by dynamically generating and executing read and write queries via the ORM's (*i.e.,* Diesel's) DSL API. Next, let us see how we refine the API to implement RDiesel which precisely tracks query semantics in a modular, type-directed fashion.

[2]We might think of this approach as using trait machinery to implement one of the classic use-cases for Generalized Algebraic Data Types [2, 31].

Generic Refinements for `Expr` As Diesel's queries are all abstracted by the `Expr` trait, in RDiesel we track the semantics of each (implementation) of `Expr` using an associated refinement `eval` that specifies what the `Expr` evaluates to. The implementations of `Expr` for program values specify that the associated refinement equals that `val`ue. Similarly, the RDiesel implementations of `Expr` for the (generated) type-level fields, stipulate that the associated refinement `eval` returns the corresponding element of the row.

```
trait Expr<R, V> { reft eval(e: Self, r: R) -> V; }

// Implementation of Expr for program values
impl<R> Expr<R, i32> for i32 {
    reft eval(val: int, _: R) -> int { val }
}

impl<R> Expr<R, usize> for usize {
    reft eval(val: int, _: R) -> int { val }
}

// Implementation of Expr for type-level fields
impl Expr<Item, i32> for ItemOwner {
    reft eval(_: Self, item: Item) -> int {
        item.owner
    }
}

impl Expr<Item, i32> for ItemPrice {
    reft eval(_: Self, item: Item) -> int {
        item.price
    }
}
```

Generic Refinements for `Expr` Composition Next, RDiesel uses Rust's trait projection mechanism to implement the associated refinement for the composition of queries. For example, for `Gt` (resp. `Eq`) expressions, the associated refinement invokes the `eval` function for the `Expr` implementations of the two sub-expressions and checks if the first is greater than (resp. equal to) the second. Similarly, for `And` expressions, the associated refinement conjoins the results of invoking `eval` on the two sub-expressions.

```
impl<R, A:Expr<R, i32>, E2:Expr<R, i32>> Expr<R, i32> for Add<A, B> {
    reft eval(e: Self, row: R) -> V { A::eval(e.lhs, row) + B::eval(e.rhs, row) }
}
impl<R, V, A: Expr<R, V>, B: Expr<R, V>> Expr<R, bool> for Gt<V, A, B> {
    reft eval(e: Self, row: R) -> V { A::eval(e.lhs, row) > B::eval(e.rhs, row) }
}
impl<R, V, A: Expr<R, V>, B: Expr<R, V>> Expr<R, bool> for Eq<V, A, B> {
    reft eval(e: Self, row: R) -> V { A::eval(e.lhs, row) == B::eval(e.rhs, row) }
}
impl<R, A: Expr<R, bool>, B: Expr<R, bool>> Expr<R, bool> for And<A, B> {
    reft eval(e: Self, row: R) -> V { A::eval(e.lhs, row) && B::eval(e.rhs, row) }
}
```

6.2.3 Access Control Verification. Finally, let us see how RDiesel lets us statically verify that the database-backed applications manipulate sensitive user data according to application specific access control policies that govern which users can read or write the database rows and columns.

Generic Access Control for Reads Building on the refined `Expr` trait, RDiesel defines the method `select_where` to *read* all the rows of a table that satisfy a query. The method takes in a `Context<U>` carrying the database connection (the parameter `U` is discussed next) and a query `Q` implementing `Expr<R, bool>`:

```
fn select_where<U, R, Q: Expr<R, bool>>(cx: &Context<U>, qry: Q) -> Vec<R{r: Q::eval(qry, r)}>
```

We use the associated generic for the query `Q` to refine the API to specify that `select_where` only returns rows on which the query evaluates to `true`.

Generic Access Control for Updates RDiesel also provides an `update_where` method to *write* a new value into the field (column) of each row that satisfies some query

```
fn update_where<F, U, R, Q>(cx: &Context<U>, qry: Q, fld: F, val: V)
requires forall row. Q::eval(qry, row) => F::allow_update(cx.user, row);
where
    Q: Expr<R, bool>,
    F: FieldPolicy<R, U>,

trait FieldPolicy<R, U> {
    reft allow_update(user: U, row: R) -> bool;
}
```

Crucially, RDiesel refines the type of `update_where` to specify fine-grained access control policies that restrict the sets of users who are allowed to *write* to a particular field of the database. To this end, RDiesel's API requires fields to implement the `FieldPolicy` trait, which includes an associated refinement that specifies whether a given `user` is allowed to update the (corresponding field of) the given `row`. By tracking the current authenticated user in the `Context`, the `update_where` method `requires` that *every* row returned by evaluating the `query` (*i.e.,* every row that may get updated) be one that the user is allowed to update, per the `allow_update` refinement specified in the `FieldPolicy`.[3] Finally, we highlight that the API is generic on the type of the user `U` which can be instantiated to a concrete user type by client applications.

Generic Access Control for Insert Similar to updates, RDiesel defines a way to control which users are allowed to insert rows in a table. In this case, the access is controlled by the `RowPolicy` trait that must be implemented on the row type to specify when a user is allowed to insert a row via the `allow_insert` associated refinement. The `insert_row` method, uses `allow_insert` to specify that the authenticated user must be allowed to insert the row.

```
fn insert_row<F, U, R, Q>(cx: &Context<U>, row: R)
requires R::allow_insert(cx.user, row);
where
    Q: Expr<R, bool>,
    F: RowPolicy<R, U>,

trait RowPolicy<R, U> {
    reft allow_insert(user: U, row: R) -> bool;
}
```

Example: Verifying Read Policies The function `show_items` in Fig. 14 retrieves all the items that are owned by the user `owner`. The `impl Expr` for `ItemOwner` and `ItemLevel` in Fig. 14 have their associated `eval` generic refinements respectively be the functions `|_,item| item.owner` and `|_,item| item.level`. Hence, the `impl Expr` for the first `qry` on line 6 has the associated `eval` refinement `|_, item| item.owner == owner`. Therefore, the `items` returned in line 8 are guaranteed to be owned by the `viewer`. In the else branch, where the `viewer` is *not* the `owner`, the `qry` is conjoined with a clause that checks the `ItemLevel` is `PUBLIC`. That is, the `impl Expr` for the second `qry` on line 10 has the associated refinement `|_, item| item.owner == owner && item.level == PUBLIC`, and hence the `items` returned in line 12 are guaranteed to have their `level` set to `PUBLIC`. Consequently, Flux statically verifies the read access control policy that says that every item read by the `viewer` is either owned by them, or is marked `PUBLIC`.

[3]Both Diesel and RDiesel allow simultaneously updating multiple fields via `Changeset`s but we omit them for simplicity.

```
1  fn show_items(
2      cx:&Context<User>,
3      owner: usize
4  ) {
5      let viewer = cx.auth_user().id;
6      let qry = eq(ItemOwner, owner);
7      let items = if viewer == owner {
8          select_where(cx, qry)
9      } else {
10         let qry = qry.and(
11             eq(ItemLevel, PUBLIC));
12         select_where(cx, qry)
13     };
14     // READ POLICY
15     for item in items {
16         assert(item.owner == viewer ||
17                item.level == PUBLIC)
18     }
19 }
```

```
1  fn update_item_descr(
2      cx: &Context<U>,
3      item_id: ItemId,
4      descr: String
5  ) {
6      let updater = cx.auth_user().id;
7      let qry = eq(ItemId, item_id);
8      let qry_own = eq(ItemOwner, updater);
9      let qry = and(qry, qry_own);
10     update_where(cx, qry, ItemDescr, descr);
11 }
12
13
14 // UPDATE POLICY
15 impl FieldPolicy<Item, User> for ItemDescr {
16     reft allow_update(u: User, i: Item) -> bool {
17         u.id == i.owner
18     }
19 }
```

Fig. 14. Verifying read policies (L) and update policies (R) in web-applications.

Example: Verifying Update Policies Fig. 14 shows a function that updates the `descr` field of a given item using the `update_where` API call. The update policy for the `ItemDescr` field states that only the user who owns the item is allowed to update the field, *i.e.,* the owner must be the authorized user performing the field update. The `qry` on line 9 has the associated `eval` refinement `|_, item| item.id == item_id && item.owner == updater`, and hence, every row matching the above is indeed owned by the authorized user `updater`, thereby verifying the update satisfies the policy. However, if the code *omitted* the lines 8 and 9 then Flux would *reject* the code as it would allow a user other than the owner to update the item's description.

6.2.4 Implementing Web Applications with RDiesel. We ported some of the web applications from [17] to Rust using the RDiesel API and verified the same access control policies. Fig. 3 summarizes the results. The specification size is modest, most of which correspond to boilerplate required to implement the various traits. This size could be further reduced by providing a more concise syntax via a procedural macro.

7 Related Work

Interactive Verifiers based on higher order logics like Rocq [4], Isabelle [20], Lean [7], and dependently-typed languages like Agda [21], directly allow for quantifying over propositions, and hence, can easily express the same specifications as generic refinements, which, ultimately are a way to parameterize invariants or properties over a set of contracts (types). However, all these systems require the programmer to explicitly instantiate the quantifiers at each usage site (and then potentially interactively prove various facts about the instantiations).

Autoactive Verifiers like Dafny [19], F* [25], and Why3 [11], allow users to write Floyd-Hoare style specifications using loop invariants, pre- and post-conditions, and then automatically verify those using SMT solvers. F* and Why3 additionally support higher-order specifications (*i.e.,* quantifying over invariants). However, the SMT validity queries issued by these tools contain universal quantifiers, and hence, fall outside the decidable logical fragments supported by SMT

solvers. Consequently, verification is undecidable in theory, and somewhat brittle in practice, as the success of the automated verification hinges upon the particular SMT solver's unpredictable quantifier instantiation heuristics. Additionally, the higher order predicates must be explicitly instantiated, which is unpleasant for the programmer. In contrast, λ_G's generic refinements are carefully engineered to yield decidable SMT queries, and to be automatically instantiable at usage sites via a combination of unification and horn constraint solving.

Rust Verifiers like Prusti [1], Verus [15], and Creusot [9] bring SMT-based, Floyd-Hoare style verification to Rust, by cleverly exploiting Rust's ownership type system to simplify reasoning about state. Creusot [8] and Prusti [5] support modular specifications for *traits* by allowing users to write *ghost methods* that describe additional properties of traits, which can then be used in other contracts on the traits (much like our associated generic refinement `in_bounds` in Fig. 6). However, none of them support quantifying over invariants in contracts in the style of λ_G's generic refinement types. In contrast, we view generic refinements as a way to parameterize types and contracts over invariants while preserving decidable checking and instantiation. As an added bonus, generics generalize nicely to the trait setting with *typeclass-as-dictionary* interpretation as discussed in § 2.4.

Interactive Rust Verifiers like Aeneas [13] and RefinedRust[12], work by translating Rust programs into Lean or Rocq bringing the full power of interactive theorem proving to verifying Rust. Consequently, these tools can also specify arbitrary higher-order contracts about Rust functions, but this expressiveness comes at the cost of interactive proof.

Refinement Type based Verifiers like Liquid Haskell [29] support a feature called *abstract refinements* [28] that are very closely related to λ_G's generic refinements. In particular, abstract refinements are similar to λ_G's Horn generics (*i.e.,* **hrn** parameters). Horn generics have limited expressiveness as they have to appear only under top-level conjunctions which makes it impossible to use them in more complex ways, *e.g.,* under negations or disjunctions. LiquidHaskell also supports *bounded refinements* [27] which allow for various ways to relate abstract refinements, but these are significantly more complicated to write (as specifying the correct bounds can require some substantial mental gymnastics), and slower to check (as instantiating the bounds requires solving Horn constraints). In contrast, this work shows how *indices* [32] in particular, the bidirectional synthesis (instantiation) technique of Economou et al. [10] can be used as foundation upon which to build a means of quantifying over refinements, that permits simple specification, and decidable and efficient instantiation and verification.

Acknowledgments

We thank the anonymous referees for their excellent suggestions for improving the paper. This work was supported by NSF grants CNS-2327336, CNS-2155235, CNS-2120642 and generous gifts from Microsoft Research.

References

[1] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust Types for Modular Specification and Verification. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 147 (oct 2019), 30 pages. https://doi.org/10.1145/3360573

[2] Lennart Augustsson and Kent Petersson. 1994. Silly Type Families. (1994). https://web.cecs.pdx.edu/~sheard/papers/silly.pdf.

[3] Joao Filipe Belo, Michael Greenberg, Atsushi Igarashi, and Benjamin C Pierce. 2011. Polymorphic contracts. In *Programming Languages and Systems: 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 20.* Springer, 18–37. https://doi.org/10.1007/978-3-642-19718-5_2

[4] Yves Bertot and Pierre Castran. 2010. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions* (1st ed.). Springer Publishing Company, Incorporated.

[5] Aurel Bílý, Jonas Hansen, Peter Müller, and Alexander J. Summers. 2022. Compositional Reasoning for Side-effectful Iterators and Iterator Adapters. *CoRR* abs/2210.09857 (2022). https://doi.org/10.48550/ARXIV.2210.09857 arXiv:2210.09857

[6] Benjamin Cosman and Ranjit Jhala. 2017. Local Refinement Typing. *Proc. ACM Program. Lang.* 1, ICFP, Article 26 (aug 2017), 27 pages. https://doi.org/10.1145/3110270

[7] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *Automated Deduction - CADE-25*, Amy P. Felty and Aart Middeldorp (Eds.). Springer International Publishing, Cham, 378–388. https://doi.org/10.1007/978-3-319-21401-6_26

[8] Xavier Denis and Jacques-Henri Jourdan. 2023. Specifying and Verifying Higher-order Rust Iterators. In *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 13994)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer, 93–110. https://doi.org/10.1007/978-3-031-30820-8_9

[9] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering*, Adrian Riesco and Min Zhang (Eds.). Springer International Publishing, Cham, 90–105. https://doi.org/10.1007/978-3-031-17244-1_6

[10] Dimitrios J. Economou, Neel Krishnaswami, and Jana Dunfield. 2023. Focusing on Refinement Typing. *ACM Trans. Program. Lang. Syst.* 45, 4, Article 22 (dec 2023), 62 pages. https://doi.org/10.1145/3610408

[11] Jean-Christophe Filliâtre and Andrei Paskevich. 2013. Why3 - Where Programs Meet Provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 125–128. https://doi.org/10.1007/978-3-642-37036-6_8

[12] Lennard Gaher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 2023. Refined Rust: Towards high-assurance verification of unsafe Rust programs. https://people.mpi-sws.org/~gaeher/slides/refinedrust_rw23.pdf

[13] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust Verification by Functional Translation. *Proc. ACM Program. Lang.* 6, ICFP, Article 116 (aug 2022), 31 pages. https://doi.org/10.1145/3547647

[14] Ranjit Jhala and Niki Vazou. 2021. Refinement Types: A Tutorial. *Foundations and Trends® in Programming Languages* 6, 3–4 (2021), 159–317. https://doi.org/10.1561/2500000032

[15] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types. *Proc. ACM Program. Lang.* 7, OOPSLA1 (2023), 286–315. https://doi.org/10.1145/3586037

[16] Nico Lehmann, Adam T. Geller, Niki Vazou, and Ranjit Jhala. 2023. Flux: Liquid Types for Rust. *Proc. ACM Program. Lang.* 7, PLDI, Article 169 (jun 2023), 25 pages. https://doi.org/10.1145/3591283

[17] Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. 2021. STORM: Refinement Types for Secure Web Applications. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 441–459. https://www.usenix.org/conference/osdi21/presentation/lehmann

[18] Nico Lehmann, Cole Kurashige, Nikhil Akiti, Niroop Krishnakumar, and Ranjit Jhala. 2025. Generic Refinement Types - Technical Appendix. https://github.com/flux-rs/popl25

[19] K. Rustan M. Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. https://doi.org/10.1007/978-3-642-17511-4_20

[20] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. https://link.springer.com/book/10.1007/3-540-45949-9

[21] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph. D. Dissertation. Chalmers.

[22] Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (jan 2000), 1–44. https://doi.org/10.1145/345099.345100

[23] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

[24] Taro Sekiyama, Atsushi Igarashi, and Michael Greenberg. 2017. Polymorphic manifest contracts, revised and resolved. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39, 1 (2017), 1–36. https://doi.org/10.1145/2994594

[25] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2011. Secure distributed programming with value-dependent types. In *Proceeding of the 16th ACM SIGPLAN international conference on Functional Programming, ICFP 2011, Tokyo, Japan, September 19-21, 2011*, Manuel M. T. Chakravarty, Zhenjiang Hu,

and Olivier Danvy (Eds.). ACM, 266–278. https://doi.org/10.1145/2034773.2034811
[26] The Diesel Core Team. 2024. Diesel: A Safe, Extensible ORM and Query Builder for Rust. (2024). https://diesel.rs.
[27] Niki Vazou, Alexander Bakst, and Ranjit Jhala. 2015. Bounded refinement types. In *ICFP*. https://doi.org/10.1145/2784731.2784745
[28] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7792)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer, 209–228. https://doi.org/10.1007/978-3-642-37036-6_13
[29] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 269–282. https://doi.org/10.1145/2628136.2628161
[30] P. Wadler and S. Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '89)*. Association for Computing Machinery, New York, NY, USA, 60–76. https://doi.org/10.1145/75277.75283
[31] Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150
[32] Hongwei Xi and Frank Pfenning. 1998. Eliminating array bound checking through dependent types. *SIGPLAN Not.* 33, 5 (may 1998), 249–257. https://doi.org/10.1145/277652.277732
[33] Christoph Zenger. 1997. Indexed types. *Theor. Comput. Sci.* 187, 1–2 (nov 1997), 147–165. https://doi.org/10.1016/S0304-3975(97)00062-5

Received 2024-07-11; accepted 2024-11-07