Linearly Qualified Types

Generic Inference for Capabilities and Uniqueness

ARNAUD SPIWACK, Tweag, France
CSONGOR KISS, Imperial College London, United Kingdom
JEAN-PHILIPPE BERNARDY, University of Gothenburg, Sweden
NICOLAS WU, Imperial College London, United Kingdom
RICHARD A. EISENBERG, Tweag, France

A linear parameter must be consumed exactly once in the body of its function. When declaring resources such as file handles and manually managed memory as linear arguments, a linear type system can verify that these resources are used safely. However, writing code with explicit linear arguments requires bureaucracy. This paper presents *linear constraints*, a front-end feature for linear typing that decreases the bureaucracy of working with linear types. Linear constraints are implicit linear arguments that are filled in automatically by the compiler. We present linear constraints as a qualified type system, together with an inference algorithm which extends GHC's existing constraint solver algorithm. Soundness of linear constraints is ensured by the fact that they desugar into Linear Haskell.

CCS Concepts: • **Software and its engineering → Language features**; *Functional languages*; *Formal language definitions*.

Additional Key Words and Phrases: GHC, Haskell, linear logic, linear types, constraints, qualified types, inference

ACM Reference Format:
Arnaud Spiwack, Csongor Kiss, Jean-Philippe Bernardy, Nicolas Wu, and Richard A. Eisenberg. 2022. Linearly Qualified Types: Generic Inference for Capabilities and Uniqueness. *Proc. ACM Program. Lang.* 6, ICFP, Article 95 (August 2022), 28 pages. https://doi.org/10.1145/3547626

1 INTRODUCTION

Linear type systems have seen a renaissance in recent years in various programming communities. Rust's ownership system guarantees memory safety for systems programmers, Haskell's GHC 9.0 includes support for linear types, and even dependently typed programmers can now use linear types with Idris 2. All of these systems are vastly different in ergonomics and scope. Rust uses dedicated syntax and code generation to support management of resources, while Linear Haskell is a type system change without any other impact on the compiler, such as in the code generator or runtime system. Linear Haskell is designed to be general purpose, but using its linear arguments to emulate Rust's ownership model is a tedious exercise. It requires the programmer to carefully thread resource tokens.

Authors' addresses: Arnaud Spiwack, Tweag, Paris, France, arnaud.spiwack@tweag.io; Csongor Kiss, Imperial College London, London, United Kingdom, csongor.kiss14@imperial.ac.uk; Jean-Philippe Bernardy, University of Gothenburg, Gothenburg, Sweden, jean-philippe.bernardy@gu.se; Nicolas Wu, Imperial College London, London, United Kingdom, n.wu@imperial.ac.uk; Richard A. Eisenberg, Tweag, Paris, France, rae@richarde.dev.

2475-1421/2022/8-ART95
https://doi.org/10.1145/3547626

To get a sense of the power and the tedium of programming with linear types, consider the following function:

```
read2AndDiscard :: MArray a ⊸ (Ur a, Ur a)
read2AndDiscard arr0 =
  let (arr1, x) = read arr0 0
      (arr2, y) = read arr1 1
      ()        = free arr2
  in (x, y)
```

This function reads the first two elements of an array and returns them after deallocating the array. Linearity enables the array library to ensure that there is only one reference to the array, and therefore it can be mutated in-place without violating referential transparency. Let us stress that this uniqueness property is an invariant of the array library, not an intrinsic property of linear functions.

After the array has been freed, it is no longer possible to read or write to it. Notice that the *read* function consumes the array and returns a fresh array, to be used in future operations. Operationally, the array remains the same, but each operation assigns a new name to it, thus facilitating tracking references statically. Finally, *free* consumes the array without returning a new one, statically guaranteeing that it can no longer be used. The values x and y read from the array are returned; their types include elements wrapped by the *Ur* (pronounced "unrestricted", and corresponding to the "!" operator of Girard [1987]) type, allowing them to be used arbitrarily many times. This works because *read2AndDiscard* takes a restricted-use array containing unrestricted elements. In a non-linear language, one would have to forgo referential transparency to handle mutable operations either by using a monadic interface or allowing arbitrary effects. Compare the above function with what one would write in a non-linear, impure language:

```
read2AndDiscard :: MArray a → (a, a)
read2AndDiscard arr =
  let x  = read arr 0
      y  = read arr 1
      () = unsafeFree arr
  in (x, y)
```

This non-linear version does not guarantee that there is a unique reference to the array, so freeing the array is a potentially unsafe operation. However, it is simpler because there is less bureaucracy to manage: we are clearly interacting with the *same* array throughout, and this version makes that apparent. We see here a clear tension between extra safety and clarity of code—one we wish, as language designers, to avoid. How can we get the compiler to see that the array is used safely without explicit threading?

Following well-known ideas [Crary et al. 1999; Smith et al. 2000; Walker and Morrisett 2000], our approach is to let arrays be unrestricted, but associate linear capabilities (such as *Read*, *Write*) to them. In fact, we show in this paper that such linear capabilities are the natural analogue of Haskell's type class constraints to the setting of linear types. We call these new constraints *linear constraints*. Like class constraints, linear constraints are propagated implicitly by the compiler. Like linear arguments, they can safely be used to track resources such as arrays or file handles. Thus, linear constraints are the combination of these two concepts, which have been studied independently elsewhere [Bernardy et al. 2017; Cervesato et al. 2000; Hodas and Miller 1994; Vytiniotis et al. 2011].

With our extension, we can write a new pure version of *read2AndDiscard* which does not require explicit threading of the array:

```
read2AndDiscard :: (Read n, Write n) ⊸ UArray a n → (Ur a, Ur a)
read2AndDiscard arr =
  let □x  = read arr 0
      □y  = read arr 1
      □() = free arr
  in (x, y)
```

The only changes from the impure version are that this version explicitly requires having read and write access to the array, and pattern-matching against □ (read "pack") is necessary in order to access the linear constraint packed in the result of *read* and *free*. (Section 8.3 suggests how we can get rid of □, too.) Crucially, the resource representing the ownership of the array is a linear constraint and is separate from the array itself, which no longer needs to be threaded manually.

Our contributions are as follows:

- A system of qualified types that allows a constraint assumption to be given a multiplicity (linear or unrestricted). Linear assumptions are used precisely once in the body of a definition (Section 5). This system supports examples that have motivated the design of several resource-aware systems, such as ownership *à la* Rust (Section 4), or capabilities in the style of Mezzo [Pottier and Protzenko 2013] or ATS [Zhu and Xi 2005]; accordingly, our system points towards a possible unification of these lines of research.
- Applications of this qualified type system to allow writing
  - resource-aware algorithms without explicit threading (Section 4); and
  - functions whose result can only be used linearly (Section 3.2)
- An inference algorithm that respects the multiplicity of assumptions. We prove that this algorithm is sound with respect to our type system (Section 6). It consists of
  - a constraint generation algorithm (Section 6.2). The language of generated constraints tracks multiplicities.
  - a solver (Section 6.3) for the generated constraints, which restricts proof-search algorithms for linear logic in order to be *guess free* [Vytiniotis et al. 2011, Section 6.4]. A guess-free algorithm ensures that constraint inference is predictable and insensitive to small changes in the source program; it is necessarily incomplete.

Our language is given semantics by desugaring into a core language based on that of Bernardy et al. [2017]. Our design is intended to work well with other features of Haskell and GHC extensions. Indeed, we have a prototype implementation (Section 8.1).

2 BACKGROUND: LINEAR HASKELL

This section, mostly cribbed from Bernardy et al. [2017, Section 2.1], describes our baseline approach, as released in GHC 9.0. Linear Haskell adds a new type of functions, dubbed *linear functions*, and written $a \multimap b$.[1] A linear function consumes its argument exactly once. Linear Haskell defines it as follows:

> $f :: a \multimap b$ guarantees that if $(f\ u)$ is consumed exactly once, then the argument u is consumed exactly once.

[1]The linear function type and its notation come from linear logic [Girard 1987], to which the phrase *linear types* refers. All the various design of linear typing in the literature amount to adding such a linear function type, but details can vary wildly. See Bernardy et al. [2017, Section 6] for an analysis of alternative approaches.

To make sense of this statement we need to know what "consumed exactly once" means. Our definition is based on the type of the value concerned:

Definition 2.1 (Consume exactly once).

- To consume a value of atomic base type (like *Int*) exactly once, just evaluate it.
- To consume a function exactly once, apply it to one argument, and then consume its result exactly once.
- To consume a pair exactly once, pattern-match on it, and then consume each component exactly once.
- In general, to consume a value of an algebraic datatype exactly once, pattern-match on it, and then consume all its linear components exactly once.

2.1 Multiplicities

The usual arrow type $a \to b$ can be recovered using Ur, as $Ur\ a \multimap b$, but Linear Haskell provides a first-class treatment of $a \to b$, thus ensuring backwards compatibility with Haskell. In practice, the type-checker records the *multiplicity* of every introduced variable: 1 for linear arguments and ω for unrestricted ones. This way, one can give a unified treatment of both arrow types [Kobayashi et al. 1999].

$$\pi, \rho \quad ::= \quad 1 \mid \omega \qquad \text{Multiplicities}$$

We stress that a multiplicity of 1 restricts *how the variable can be used*. It does not restrict *which values can be substituted for it*. In particular, a linear function cannot assume that it is given the unique pointer to its argument. For example, if $f :: a \multimap b$, then the following is fine:

$g :: a \to (b, b)$
$g\ x = (f\ x, f\ x)$

The type of g makes no guarantees about how it uses x. In particular, g can pass x to f.

Pattern matching on a value of type $Ur\ \ a$ yields a payload of multiplicity ω, even when the scrutinee has multiplicity 1. In general, given a multiplicity set, the desired (sub)structural rules can be obtained by endowing multiplicities with the appropriate semiring structure [Abel and Bernardy 2020]. In this paper, we use the same multiplicity structure as Linear Haskell:[2,3]

$$\left\{ \begin{array}{lcl} \pi + \rho & = & \omega \end{array} \right. \qquad \left\{ \begin{array}{lcl} 1 \cdot \pi & = & \pi \\ \omega \cdot \pi & = & \omega \end{array} \right.$$

2.2 Shortcomings of Linear Haskell that We Address

The *read* function in Section 1 consumes the array it operates on. Therefore, the same array can no longer be used in further operations: doing so would result in a type error. To resolve this, a new name for the same array is produced by each operation.

From the perspective of the programmer, this is unwanted boilerplate. Minimizing such boilerplate is the main aim of this paper. Our approach is to let the array be non-linear, and let its

[2]Even though linear Haskell additionally supports multiplicity polymorphism, we do not support multiplicity polymorphism on constraint arguments. Linear Haskell takes advantage of multiplicity polymorphism to avoid duplication of higher-order functions. The prototypical example is $map :: (a \to_m b) \to [a] \to_m [b]$, where $\to_m$ is the notation for a function arrow of multiplicity m. First-order functions, on the other hand, do not need multiplicity polymorphism, because linear functions can be η-expanded into unrestricted functions as explained in Section 2. Higher-order functions whose arguments are themselves constrained functions are rare, so we do not yet see the need to extend multiplicity polymorphism to apply to constraints. Futhermore, it is not clear how to extend the constraint solver of Section 6.3 to support multiplicity-polymorphic constraints.

[3]Here and in the rest of the paper we adopt the convention that equations defining a function by pattern matching are marked with a { to their left.

$new :: Int \to (MArray\ a \multimap Ur\ r) \multimap Ur\ r$
$write :: MArray\ a \multimap Int \to a \to MArray\ a$
$read :: MArray\ a \multimap Int \to (MArray\ a, Ur\ a)$
$free :: MArray\ a \multimap ()$

(a) Linear Types

type $RW\ n = (Read\ n, Write\ n)$
$new :: Linearly \mathrel{=\!\!\circ} Int \to \exists\ n.UArray\ a\ n \otimes RW\ n$
$write :: RW\ n \mathrel{=\!\!\circ} UArray\ a\ n \to Int \to a \to () \otimes RW\ n$
$read :: Read\ n \mathrel{=\!\!\circ} UArray\ a\ n \to Int \to Ur\ a \otimes Read\ n$
$free :: RW\ n \mathrel{=\!\!\circ} UArray\ a\ n \to ()$

(b) Linear Constraints

Fig. 1. Interfaces for mutable arrays

capabilities (*i.e.* having read or write access) be *linear constraints*. Once these capabilities are consumed, the array can no longer be read from or written to without triggering a compile time error.

A further drawback of today's Linear Haskell is that the programmer cannot restrict how a linear function is used. For example, suppose we want to use linear types to create a pure interface to arrays that supports in-place mutation; the interface is safe only if we guarantee that arrays cannot be aliased. Because the result of a hypothetical *newArray* function can be stored in an unrestricted variable (of multiplicity ω), the linearity system cannot prevent its aliasing. Instead, Bernardy et al. [2017, Fig. 2] use a continuation-passing style to enforce non-aliasing.

3 WORKING WITH LINEAR CONSTRAINTS

Consider the Haskell function *show*:

$show :: Show\ a \Rightarrow a \to String$

In addition to the function arrow $\to$, common to all functional programming languages, the type of this function features a constraint arrow $\Rightarrow$. Everything to the left of a constraint arrow is called a *constraint*, and will be highlighted in blue throughout the paper. Here $Show\ a$ is a class constraint.

Constraints are handled implicitly by the typechecker. That is, if we want to *show* the integer $n :: Int$ we would write *show n*, and the typechecker is responsible for proving that $Show\ Int$ holds, without intervention from the programmer.

For our *read2AndDiscard* example, the $(Read\ n, Write\ n)$ (abbreviated as $RW\ n$) constraint represents read and write access to the array tagged with the type variable *n*. (The full API under consideration appears in Fig. 1b.) That is, the constraint $RW\ n$ is provable if and only if the array tagged with *n* is readable and writable. This constraint is linear: it must be consumed (that is, used as an assumption in a function call) exactly once. In order to manage linearity implicitly, this paper introduces a linear constraint arrow ($\mathrel{=\!\!\circ}$), much like Linear Haskell introduces a linear function arrow ($\multimap$). Constraints to the left of a linear constraint arrow are *linear constraints*. Using the linear constraint $RW\ n$, we can give the following type to *free*:

$free :: RW\ n \mathrel{=\!\!\circ} UArray\ a\ n \to ()$

There are a few things to notice:

- We have introduced a new type variable *n*. In contrast, the version in Figure 1a without linear constraints has type $free :: MArray\ a \multimap ()$. The type variable *n* is a type-level tag used to identify the array.
- The run-time variable representing the array can now be used multiple times. Instead of restricting the use of this variable, the linear constraint $RW\ n$ controls access to the array.
- If we have a single, linear, $RW\ n$ available, then after *free* there will not be any $RW\ n$ left to use, thus preventing the array from being used after freeing. This is precisely what we were trying to achieve.

The above deals with freeing an array and ensuring that it cannot be used afterwards. However, we still need to explain how a constraint $RW\ n$ can come into scope. The type of *new* with linear constraints is as follows:

$$new :: Linearly \multimap Int \to \exists\ n.UArray\ a\ n \otimes RW\ n$$

This type, too, illustrates several new aspects:

- The *Linearly* constraint is a linear constraint, though it takes no type parameter. It restricts the result of *new* to be used linearly, meaning that any variable that stores the result of *new* must have multiplicity 1. *Linearly* is explained more fully in Section 3.2.
- Because *UArray* is now parameterised by a type-level tag *n*, *new* must return a *UArray* with a fresh such *n*. This is achieved through returning an existentially-quantified type[4] packing the type variable *n*. Such types are introduced with the □ constructor.
- Not only do we need a fresh type variable *n*, but we also need to introduce the linear constraint $RW\ n$ for use in subsequent calls to *read* and *free*. Our existentials also allow packing a constraint, thanks to the $\otimes$ operator.

With all these features working together, we see that *new* returns a non-duplicable *UArray* tagged with *n*, accessible only when the $RW\ n$ constraint is available.

We must also ensure that *read* can both promise to operate only on a readable array and that the array remains readable afterwards. That is, *read* must both consume a linear constraint $Read\ n$ and also produce a fresh linear constraint $Read\ n$, as we see in Fig. 1b, and repeated here:

$$read :: Read\ n \multimap UArray\ a\ n \to Int \to Ur\ a \otimes Read\ n$$

We have now seen all the ingredients needed to write the *read2AndDiscard* example as in Section 1.

3.1 Minimal Examples

To get a sense of how the features that we introduce should behave, we now look at some simple examples. Using constraints to represent limited resources allows the typechecker to reject certain classes of ill-behaved programs. Accordingly, the following examples show the different reasons a program might be rejected.

In what follows, we will be using a constraint C that is consumed by the *useC* function.

$$useC :: C \multimap Int$$

The type of *useC* indicates that it consumes the linear resource C exactly once.

3.1.1 Dithering. We reject this program:

```
dithering :: C ⊸ Bool → Int
dithering x = if x then useC else 10
```

The problem with *dithering* is that it does not unconditionally consume C: the branch where $x \equiv True$ uses the resource C, whereas the other branch does not.

[4]There is a variety of ways that existential types can be worked into a language. The existentials that we use here may be understood as a generalisation of those presented by Pierce [2002, Chapter 24]. However, Eisenberg et al. [2021] work out an approach that makes linear constraints even easier to use, as we discuss in Section 8.3. In order to separate concerns, we do not build our formalism on Eisenberg et al. [2021], instead modeling our existentials on the more widely known formulation described by Pierce [2002]. We additionally freely omit the $\exists\ a_1 \ldots a_n.$ or $\otimes Q$ parts when they are empty. The idea of using existentials to return linear capabilities can be attributed to Fluet et al. [2006].

3.1.2 Neglecting. Now consider the type of the linear version of *const*:

$const :: a \multimap b \to a$

This function uses its first argument linearly, and ignores the second. Thus, the the second arrow is unrestricted. One way to improperly use the linear *const* is by neglecting a linear variable:

$neglecting :: C \Rightarrow Int$
$neglecting = const\ 10\ useC$

The problem with *neglecting* is that, although *useC* is mentioned in this program, it is never consumed: *const* does not use its second argument. The constraint C is not consumed exactly once, and thus this program is rejected. The rule is that a linear constraint can only be consumed (linearly) in a linear context. For example,

$notNeglecting :: C \Rightarrow Int$
$notNeglecting = const\ useC\ 10$

is accepted, because the C constraint is passed on to *useC* which itself appears as an argument to a linear function (whose result is itself consumed linearly).

3.1.3 Overusing. Finally, the following program is rejected because it uses C twice:

$overusing :: C \Rightarrow (Int, Int)$
$overusing = (useC, useC)$

3.2 Restricting to a Linear Context with *Linearly*

A linear function makes a promise about how it is going to use its argument, but linearity imposes no restrictions on how a function – or its result – is going to be used. The caller may use the linear function's result unrestrictedly. This poses a challenge for providing a type-safe interface for libraries that rely on having a unique pointer to some resource, such as safe mutable arrays, because the obvious definition of a constructor function can immediately be misused, violating the assumption of uniqueness.

$new :: Int \to MArray\ a$

$bad = \mathbf{let}\ arr = new\ 5\ \mathbf{in}\ (arr, arr)$
$badToo = Ur\ (new\ 5)$

However, with linear constraints, we can overcome this problem by putting the special *Linearly* constraint on *new*:

$new :: Linearly \Rightarrow Int \to MArray\ a$

Suppose we have assumed the *Linearly* constraint linearly; that is, we must use the *Linearly* assumption precisely once. Now, our definition for *bad* is rejected: either we infer *arr* to have multiplicity ω, in which case its definition uses *Linearly* ω times; or we infer *arr* to have multiplicity 1, in which case its use (twice) violates the linearity restriction. Likewise, the use of *Ur* in *badToo* requires using the *Linearly* assumption ω times.

This is promising so far, but several problems remain:

Duplicating *Linearly* What if we want to create multiple arrays, each of which having a unique pointer? If *Linearly* is assumed linearly, then $\mathbf{let}\ arr1 = new\ 5; arr2 = new\ 6$ will fail, as it uses our *Linearly* assumption twice. We thus stipulate that *Linearly* must itself be duplicable: from one assumption of *Linearly*, we must be able to satisfy any arbitrary fixed number of demands on that constraint. By "arbitrary fixed number", we mean to say that

we can duplicate *Linearly* a finite number of times, but we may not use an assumption of *Linearly* with multiplicity 1 to satisfy *Linearly* at multiplicity ω.

Discarding *Linearly* Similarly to allowing duplication, we must allow discarding, in case a function allocates no arrays at all. Accordingly, we allow a linear assumption of *Linearly* to be accepted even if the constraint is never used.

Initial assumption of *Linearly* For this approach to work, we must have an assumption of *Linearly* of multiplicity 1. We can achieve this via the following primitive:

$$linearly :: (Linearly \Rightarrow_{\circ} Ur\ r) \multimap Ur\ r$$

The argument to *linearly* will be a continuation that assumes *Linearly* with multiplicity 1. Because *linearly* returns an unrestricted value, no restricted values from the continuation can escape the scope of the *Linearly* assumption. Thus, the continuation has exactly the condition we need: a linear assumption of *Linearly*.

The pattern of using a continuation in *linearly* mirrors the use of that technique by Bernardy et al. [2017, Fig. 2]. But *linearly* is, now, the only place where we need a continuation: once we have our linear *Linearly* assumption, we can use it to produce new values that must be unique.

With just these simple ingredients – a duplicable, discardable constraint that can be assumed linearly – we can write APIs that require uniqueness without heavy use of continuations.

4 APPLICATION: MEMORY OWNERSHIP

Let us now turn back to the more substantial example introduced in Section 1: manual memory management. In functional programming languages like Haskell, memory deallocation is normally the responsibility of a garbage collector. However, garbage collection is not always desirable, either due to its (unpredictable) runtime costs, or because pointers exist between separately-managed memory spaces (for example when calling foreign functions [Domínguez 2020]). In either case, one must then resort to explicit memory allocation and deallocation. This task is error prone: one can easily forget a deallocation (causing a memory leak) or deallocate several times (corrupting data). In this section we show how to build a memory management API as a *library* using linear constraints. The library is a generalisation of the array library introduced in Section 1.

4.1 Capability Constraints

Our approach, inspired by Rust, is to represent *ownership* of a memory location, and more specifically, whether the reference is mutable or read-only. We use the linear constraints *Read n* and *Write n*, guarding read access and write access to a reference respectively. Because of linearity, these constraints must be consumed, so the API can guarantee that memory is deallocated correctly. In *Read n*, *n* is a type variable (of a special kind *Location*) which represents a memory location. Locations mediate the relationship between references and ownership constraints.

class *Read* (*n* :: *Location*) **class** *Write* (*n* :: *Location*)

To ensure referential transparency, writes can be done only when we are sure that no other part of the program has read access to the reference. Therefore, writing also requires the read capability. Thus we systematically use *RW n*, pairing both the read and write capabilities.

With these components in place, we can provide an API for mutable references.

data *AtomRef* (*a* :: *Type*) (*n* :: *Location*)

The type *AtomRef* is the type of references to values of a type *a* at location *n*. Allocation of a reference can be done using the following function.

$$newRef :: Linearly \Rightarrow_{\circ} \exists\ n.AtomRef\ a\ n \otimes RW\ n$$

The function *newRef* creates a new atomic reference, initialised with ⊥; we could also pass in an initial value, but doing so in the more general case below would add complication and obscure our main goal of demonstrating linear constraints.

To read a reference, a *Read* constraint is demanded, and then returned back. Writing is similar.

$$readRef :: Read\ n \multimap AtomRef\ a\ n \to Ur\ a \otimes Read\ n$$
$$writeRef :: RW\ n \multimap AtomRef\ a\ n \to a \to () \otimes RW\ n$$

Note that the above primitives do not need to explicitly declare effects in terms of a monad or another higher-order effect-tracking device: because the $RW\ n$ constraint is linear, passing it suffices to ensure proper sequencing of effects concerning location n.

Also note that *readRef* returns an unrestricted *copy* of the element, and *writeRef* *copies* an unrestricted element into the location. This means that while *AtomRef*s are mutable, their contents are always immutable structures.

Since there is a unique $RW\ n$ constraint per reference, we can also use it to represent ownership of the reference: access to $RW\ n$ represents responsibility (and obligation) to deallocate n:

$$freeRef :: RW\ n \multimap AtomRef\ a\ n \to ()$$

4.2 Arrays

The above toolkit handles references to base types just fine. But what about storing references in objects managed by the ownership system? In Section 1, we presented an interface for mutable arrays whose contents are themselves immutable. Our approach scales beyond that use case, supporting arrays of references, including arrays of (mutable) arrays.

$$\textbf{data}\ PArray\ (a :: Location \to Type)\ (n :: Location)$$
$$newPArray :: Linearly \multimap Int \to \exists\ n.PArray\ a\ n \otimes RW\ n$$

For this purpose we introduce the type $PArray\ a\ n$, where the kind of a is $Location \to Type$: this way we can easily enforce that each reference in the array refers to the same location n. Both types $AtomRef\ a$ and $PArray\ a$ have kind $Location \to Type$, and therefore one can allocate, and manipulate arrays of arrays with this API. For example, an array of integers has type $PArray\ (AtomRef\ Int)\ n$, and indeed, the *UArray* type from Section 1 is a synonym for an array of atomic references. An array of arrays of integers would has type $PArray\ (PArray\ (AtomRef\ Int))\ n$. Thus, the framework handles nested mutable structures without any additional difficulty.

The actual runtime value of a *PArray* is a pointer to a contiguous block of memory together with the size of the memory block. This means that the length of the array can be accessed without having ownership of the array: $length :: PArray\ a\ n \to Int$. While the *PArray* reference itself is managed by the garbage collector, the pointer it contains points to manually managed memory.

4.2.1 Borrowing. The $lendMut\ arr\ i\ k$ primitive lends access to the reference at index i in arr, to a continuation function k (in Rust terminology, the function *borrows* an element of the array). Note that the continuation must return the read-write capability, so that the ownership transfer is indeed temporary. The type system guarantees that the borrowed reference cannot be shared or deallocated. Indeed, with this API, $RW\ n$ and $RW\ p$ are never simultaneously available.

$$lendMut :: RW\ n \multimap PArray\ a\ n \to Int \to (\forall\ p.\ RW\ p \multimap a\ p \to r \otimes RW\ p) \multimap r \otimes RW\ n$$

Because the elements of an array can be mutable structures (such as other arrays), reading can be done safely only if we can ensure that no one else has access to the array while the element is accessed. Otherwise, the array – including the element being read – could be mutated. Therefore, gaining simple read access to an element needs to be done using a scoped API as well:

```
swap :: RW n ⊸ PArray AtomRef n → Int → Int → () ⊗ RW n
swap arr i j | i ≡ j = □()
             | i > j = swap arr j i
             | i < j = let □(Ur (l, r)) = split arr (i + 1)
                           □()          = lendMut l i (λa_i →
                                            let □() = lendMut r (j − (i + 1)) (λa_j →
                                              let □(Ur a_i_val) = readRef a_i
                                                  □(Ur a_j_val) = readRef a_j
                                                  □()           = writeRef a_j a_i_val
                                                  □()           = writeRef a_i a_j_val
                                              in □()) in □())
                           □(Ur _)      = join l r
                       in □()
```

Fig. 2. Swapping two elements of an array

$$lend :: Read\ n \multimap PArray\ a\ n \to Int \to (\forall\ p.\ Read\ p \multimap a\ p \to r \otimes Read\ p) \multimap r \otimes Read\ n$$

For the special case of *UArray*s, a more traditional reading operation can be implemented, by lending the reference to *readRef* which creates an unrestricted *copy* of the value. This copy is under control of the garbage collector, and can escape the scope of the borrowing freely.

```
read :: Read n ⊸ UArray a n → Int → Ur a ⊗ Read n
read arr i = lend arr i readRef
```

4.2.2 Slices. It is also possible to give a safe interface to array *slices*. A slice represents a part of an array and allows splitting the ownership of the array into multiple parts, shared between different consumers. The ownership system means that slicing does not require copying.

Splitting consumes all capabilities of an array and returns two new arrays that represent the contiguous blocks of memory before and starting at a given index.

$$split :: RW\ n \multimap PArray\ a\ n \to Int \to \exists\ l\ r.Ur\ (PArray\ a\ l, PArray\ a\ r) \otimes (RW\ l, RW\ r, Slices\ n\ l\ r)$$

In addition to the array capabilities, the output constraints also include $Slices\ n\ l\ r$, witnessing the fact that locations l and r are components of n, so that they can be joined back together:

$$join :: (Slices\ n\ l\ r, RW\ r, RW\ l) \multimap PArray\ a\ l \to PArray\ a\ r \to Ur\ (PArray\ a\ n) \otimes RW\ n$$

With these building blocks, we can now implement various utility functions on arrays, such as swapping two elements of an array, which is shown in Figure 2. It is not so simple to implement[5], because we need two elements of an array simultaneously, but only one element can be borrowed at a time. To solve this problem, we split the array into two slices such that the two indices fall in two different slices. Then simply borrow the element i from the first slice, and j from the second slice (using *lendMut*). Finally, we join the two slices back together.

4.2.3 In-Place Quicksort. As an example of using the machinery defined above, we implement an in-place, pure quicksort algorithm, given in Figure 3. The *partition* function is responsible for picking a pivot element and reorganising the array elements such that each element preceding the pivot will be less than or equal to it, and the elements after will be greater than the pivot. Once

[5]Indeed, Rust's implementation uses an *unsafe* block.

```
sort :: RW n ⊸ UArray Int n → () ⊗ RW n
sort arr = let len = length arr in
  if len ⩽ 1 then □()
  else let □pivotIdx    = partition arr
           □(Ur (l, r)) = split arr pivotIdx
           □()          = sort l
           □()          = sort r
           □(Ur _)      = join l r
       in □()
```

```
partition :: RW n ⊸ UArray Int n → Int ⊗ RW n
partition arr =
  let last          = length arr − 1
      □(Ur pivot) = read arr last
      go :: RW n ⊸ Int → Int → Int ⊗ RW n
      go l r
        | l > r
        = let □() = swap arr last l in □l
        | otherwise
        = let □(Ur lVal) = read arr l in
          if lVal > pivot
          then let □() = swap arr l r
               in go l (r − 1)
          else go (l + 1) r
  in go 0 (last − 1)
```

Fig. 3. In-place quicksort

finished, it returns the index of the pivot element; *sort* then splits the array at the pivot element and recursively operates on the two slices.

5 A QUALIFIED TYPE SYSTEM FOR LINEAR CONSTRAINTS

We now present our design for a qualified type system [Jones 1994] that supports linear constraints. Our design, based on the work of Vytiniotis et al. [2011], is compatible with Haskell and GHC.

5.1 Simple Constraints and Entailment

We call constraints such as $Read\ n$ or $Write\ n$ *atomic constraints*. The set of atomic constraints is a parameter of our qualified type system.

Definition 5.1 (Atomic constraints). The qualified type system is parameterised by a set, whose elements are called *atomic constraints*. We use the variable q to denote atomic constraints.

Atomic constraints are assembled into *simple constraints* Q, which play the hybrid role of constraint contexts and (linear) logic formulae. The following operations work with simple constraints:

Scaled atomic constraints $\pi \cdot q$ is a simple constraint, where π specifies whether q is to be used linearly or not.

Conjunction Two simple constraints can be paired up $Q_1 \otimes Q_2$. Semantically, this corresponds to the multiplicative conjunction of linear logic. Tensor products represent pairs of constraints such as $(Read\ n, Write\ n)$ from Haskell.

Empty conjunction Finally we need a neutral element ε to the tensor product. The empty conjunction is used to represent functions which don't require any constraints.

However, we do not define Q inductively, because we require certain equalities to hold:

$$Q_1 \otimes Q_2 = Q_2 \otimes Q_1 \qquad \omega \cdot q \otimes \omega \cdot q = \omega \cdot q$$
$$(Q_1 \otimes Q_2) \otimes Q_3 = Q_1 \otimes (Q_2 \otimes Q_3) \qquad Q \otimes \varepsilon = Q$$

We thus say that a simple constraint is a pair combining a set of unrestricted constraints U and a multiset of linear constraints L. The linear constraints must be stored in a multiset, because assuming the same constraint twice is distinct from assuming it only once.

(1) $Q \Vdash Q$.
(2) If $Q_1 \Vdash Q_2$ and $Q \otimes Q_2 \Vdash Q_3$, then $Q \otimes Q_1 \Vdash Q_3$.
(3) If $Q \Vdash Q_1 \otimes Q_2$, then there exist Q', $Q_{\mathcal{D}}$, and Q'' such that:
$$Q_{\mathcal{D}} \in \mathcal{D}, \quad Q = Q' \otimes Q_{\mathcal{D}} \otimes Q'', \quad Q' \otimes Q_{\mathcal{D}} \Vdash Q_1, \text{ and } \quad Q_{\mathcal{D}} \otimes Q'' \Vdash Q_2.$$
(4) If $Q \Vdash \varepsilon$, then $Q \in \mathcal{D}$.
(5) If $Q_1 \Vdash Q_1'$ and $Q_2 \Vdash Q_2'$, then $Q_1 \otimes Q_2 \Vdash Q_1' \otimes Q_2'$.
(6) If $Q \Vdash \rho \cdot q$, then $\pi \cdot Q \Vdash (\pi \cdot \rho) \cdot q$.
(7) If $Q \Vdash (\pi \cdot \rho) \cdot q$, then there exists Q' such that $Q = \pi \cdot Q'$ and $Q' \Vdash \rho \cdot q$.
(8) If $Q_1 \Vdash Q_2$, then $\omega \cdot Q_1 \Vdash Q_2$.
(9) If $Q_1 \Vdash Q_2$, then for all Q', it is the case that $\omega \cdot Q' \otimes Q_1 \Vdash Q_2$.
(10) If $q \in \mathcal{D}$, then $1 \cdot q \Vdash 1 \cdot q \otimes 1 \cdot q$.
(11) If $q \in \mathcal{D}$, then $1 \cdot q \Vdash \varepsilon$.
(12) If $Q \in \mathcal{D}$ and $Q' \Vdash Q$, then $Q' \in \mathcal{D}$.

Fig. 4. Requirements for the entailment relation $Q_1 \Vdash Q_2$

Definition 5.2 (Simple constraints).

$$\begin{array}{rcll} U & ::= & \ldots & \text{set of atomic constraints } q \\ L & ::= & \ldots & \text{multiset of atomic constraints } q \\ Q & ::= & (U, L) & \text{simple constraints} \end{array}$$

We can now straightforwardly define the operations we need on simple constraints:

$$\varepsilon = (\emptyset, \emptyset) \qquad \begin{cases} 1 \cdot q = (\emptyset, q) \\ \omega \cdot q = (q, \emptyset) \end{cases} \qquad (U_1, L_1) \otimes (U_2, L_2) = (U_1 \cup U_2, L_1 \uplus L_2)$$

In practice, we do not need to concern ourselves with the concrete representation of Q as a pair of sets, instead using the operations defined just above.

The semantics of simple constraints (and, indeed, of atomic constraints) is given by an *entailment relation*. Just like the set of atomic constraints, the entailment relation is a parameter of our system.

Definition 5.3 (Entailment relation). The qualified type system is parameterised by a relation $Q_1 \Vdash Q_2$ between two simple constraints, as well as by a distinguished set $\mathcal{D}$ of duplicable atomic constraints.

We write, abusing notation, $Q \in \mathcal{D}$ for a simple constraint $Q = (U, L)$ if for all $q \in L$ we have $q \in \mathcal{D}$.

The entailment relation must obey the laws listed in Figure 4.

The set $\mathcal{D}$ is a set of constraints which can be duplicated and discarded (see Fig. 4). We use $\mathcal{D}$ to model the *Linearly* constraint. Crucially, it is *not the case* that $1 \cdot q \Vdash \omega \cdot q$ for $q \in \mathcal{D}$; such an entailment is, in fact, prohibited (Lemma 5.5) by the rules of Fig. 4. While it may seem counter-intuitive, there is nothing in linear logic mandating that a formula that can be duplicated and discarded be (equivalent to) an unrestricted formula. This observation has been exploited, for instance, to introduce so-called subexponentials [Danos et al. 1993]. For our use case, it lets the typechecker dispatch *Linearly* constraints (using duplication), but prevents the result of constrained functions to be used unrestrictedly.

An important feature of simple constraints is that, while scaling syntactically happens at the level of atomic constraints, these properties of scaling extend to scaling of arbitrary constraints. Define $\pi \cdot Q$ as:

$$
\begin{array}{llll}
a, b & \text{Type vars} & x, y & \text{Expression vars} \\
T & \text{Type constructors} & K & \text{Data constructors} \\
\sigma & ::= & \forall \overline{a}.Q \Rightarrow \tau & \text{Type schemes} \\
\tau, \upsilon & ::= & a \mid \exists \overline{a}.\tau \otimes Q \mid \tau_1 \to_\pi \tau_2 \mid T\,\overline{\tau} & \text{Types} \\
\Gamma, \Delta & ::= & \bullet \mid \Gamma, x{:}_\pi \sigma & \text{Contexts} \\
e & ::= & x \mid K \mid \lambda x.e \mid e_1\, e_2 \mid \Box\, e & \text{Expressions} \\
 & \mid & \textbf{let}\ \Box x = e_1\ \textbf{in}\ e_2 \mid \textbf{case}_\pi\ e\ \textbf{of}\ \{\overline{K_i\ \overline{x_i} \to e_i}\} & \\
 & \mid & \textbf{let}_\pi\ x = e_1\ \textbf{in}\ e_2 \mid \textbf{let}_\pi\ x : \sigma = e_1\ \textbf{in}\ e_2 &
\end{array}
$$

Context scaling $\pi \cdot \Gamma$ and addition of contexts $\Gamma_1 + \Gamma_2$ is defined as follows:

$$
\begin{cases} \pi \cdot \bullet = \bullet \\ \pi \cdot (\Gamma, x{:}_\rho \sigma) = \pi \cdot \Gamma, x{:}_{(\pi \cdot \rho)} \sigma \end{cases}
\qquad
\begin{cases} (\Gamma_1, x{:}_\pi \sigma) + \Gamma_2 = \Gamma_1 + \Gamma_2', x{:}_{(\pi+\rho)} \sigma & \text{where } \Gamma_2 = \{x{:}_\rho \sigma\} \cup \Gamma_2',\ x \notin \Gamma_2' \\ (\Gamma_1, x{:}_\pi \sigma) + \Gamma_2 = \Gamma_1 + \Gamma_2, x{:}_\pi \sigma & \text{where } x \notin \Gamma_2 \\ \bullet + \Gamma_2 = \Gamma_2 & \end{cases}
$$

Fig. 5. Grammar of the qualified type system

$$
\begin{cases} 1 \cdot (U, L) = (U, L) \\ \omega \cdot (U, L) = (U \cup L, \emptyset) \end{cases}
$$

Then the following properties hold

Lemma 5.4 (Scaling). *If $Q_1 \Vdash Q_2$, then $\pi \cdot Q_1 \Vdash \pi \cdot Q_2$.*

Lemma 5.5 (Inversion of scaling). *If $Q_1 \Vdash \pi \cdot Q_2$, then $Q_1 = \pi \cdot Q'$ and $Q' \Vdash Q_2$ for some Q'.*

Corollary 5.6 (Linear assumptions). *If $Q_1 \Vdash \omega \cdot Q_2$, then Q_1 contains no linear assumptions.*

Proofs of these lemmas (and others) appear in the extended version of the paper [Spiwack et al. 2021]; they can be proved by straightforward use of the properties in Figure 4.

5.2 Typing Rules

With this material in place, we can now present our type system. The grammar is given in Figure 5, which also includes the definitions of scaling on contexts $\pi \cdot \Gamma$ and addition of contexts $\Gamma_1 + \Gamma_2$. Note that addition on contexts is actually a partial function, as it requires that, if a variable x is bound in both Γ_1 and Γ_2, then x is assigned the same type in both (but perhaps different multiplicities). This partiality is not a problem in practice, as the required condition for combining contexts is always satisfied.

The typing rules are in Figure 6. A qualified type system [Jones 1994] such as ours introduces a judgement of the form $Q; \Gamma \vdash e : \tau$, where Γ is a standard type context, and Q is a constraint we have assumed to be true. Q behaves much like Γ, which will be instrumental for desugaring in Section 7; the main difference is that Γ is addressed explicitly, whereas Q is used implicitly in rule E-Var.

Because constraints are used implicitly, if there are several instances of the same $1 \cdot q$, it is non-deterministic which one is used in which instance of E-Var. As a consequence, we must require that any two instances of $1 \cdot q$ in a constraint Q have the same computational content (see Section 7). How do we reconcile this non-determinism with the use of linear constraints, in Section 4.2 to thread mutations? We certainly don't want type inference to non-deterministically reorder a *readRef* and a *writeRef*! The solution is that the API is arranged so that only a single instance of $RW\ n$ is ever provided. Therefore there is a single possible threading of the reads and writes. In contrast there will often be several instances of *Linearly* in scope.

$\boxed{Q; \Gamma \vdash e : \tau}$ *(Expression typing)*

E-Var
$$\frac{\Gamma_1 = x{:}_1 \forall \overline{a}. Q_1 \multimap v}{Q_1[\overline{\tau}/\overline{a}]; \Gamma_1 + \omega \cdot \Gamma_2 \vdash x : v[\overline{\tau}/\overline{a}]}$$

E-Abs
$$\frac{Q; \Gamma, x{:}_\pi \tau_1 \vdash e : \tau_2}{Q; \Gamma \vdash \lambda x.e : \tau_1 \to_\pi \tau_2}$$

E-App
$$\frac{Q_1; \Gamma_1 \vdash e_1 : \tau_1 \to_\pi \tau \qquad Q_2; \Gamma_2 \vdash e_2 : \tau_1}{Q_1 \otimes \pi \cdot Q_2; \Gamma_1 + \pi \cdot \Gamma_2 \vdash e_1\ e_2 : \tau}$$

E-Pack
$$\frac{Q; \Gamma \vdash e : \tau[\overline{v}/\overline{a}]}{Q \otimes Q_1[\overline{v}/\overline{a}]; \Gamma \vdash \Box\, e : \exists \overline{a}. \tau \otimes Q_1}$$

E-Unpack
$$\frac{Q_1; \Gamma_1 \vdash e_1 : \exists \overline{a}. \tau_1 \otimes Q \qquad \overline{a} \text{ fresh} \qquad Q_2 \otimes Q; \Gamma_2, x{:}_1 \tau_1 \vdash e_2 : \tau}{Q_1 \otimes Q_2; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ \Box x = e_1\ \mathbf{in}\ e_2 : \tau}$$

E-Let
$$\frac{Q_1 \otimes Q; \Gamma_1 \vdash e_1 : \tau_1 \qquad Q_2; \Gamma_2, x{:}_\pi Q \multimap \tau_1 \vdash e_2 : \tau}{\pi \cdot Q_1 \otimes Q_2; \pi \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}_\pi\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

E-LetSig
$$\frac{Q_1 \otimes Q; \Gamma_1 \vdash e_1 : \tau_1 \qquad \overline{a} \text{ fresh} \qquad \sigma = \forall \overline{a}. Q \multimap \tau_1 \qquad Q_2; \Gamma_2, x{:}_\pi \forall \overline{a}. Q \multimap \tau_1 \vdash e_2 : \tau}{\pi \cdot Q_1 \otimes Q_2; \pi \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}_\pi\ x : \sigma = e_1\ \mathbf{in}\ e_2 : \tau}$$

E-Case
$$\frac{Q_1; \Gamma_1 \vdash e : T\,\overline{\tau} \qquad K_i : \forall \overline{a}. \overline{v_i} \to_{\overline{\pi_i}} T\,\overline{a} \qquad Q_2; \Gamma_2, \overline{x_i{:}_{(\pi \cdot \pi_i)} v_i[\overline{\tau}/\overline{a}]} \vdash e_i : \tau}{\pi \cdot Q_1 \otimes Q_2; \pi \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_\pi\ e\ \mathbf{of}\ \{\overline{K_i\ \overline{x_i} \to e_i}\} : \tau}$$

E-Sub
$$\frac{Q_1; \Gamma \vdash e : \tau \qquad Q \Vdash Q_1}{Q; \Gamma \vdash e : \tau}$$

Fig. 6. Qualified type system

The type system of Figure 6 is purely declarative: note, for example, that rule E-App does not describe how to break the typing assumptions into constraints Q_1/Q_2 and contexts Γ_1/Γ_2. We will see how to infer constraints in Section 6. Yet, this system is our ground truth: a system with a simple enough definition that programmers can reason about typing. As is standard in the qualified-type literature (since the original paper [Jones 1994]), we do not directly give a dynamic semantics to this language; instead, we will give it meaning via desugaring to a simpler core language in Section 7.

We survey several distinctive features of our qualified type system below:

Linear functions. The type of linear functions is written $a \to_1 b$. Despite our focus on linear constraints, we still need linearity in ordinary arguments. Indeed, the linearity of arrows interacts in interesting ways with linear constraints: If $f : a \to_\omega b$ and $x : 1 \cdot q \multimap a$, then calling $f\ x$ would actually use q many times. We must make sure it is impossible to derive $1 \cdot q; f{:}_\omega a \to_\omega b, x{:}_\omega 1 \cdot q \multimap a \vdash f\ x : b$. Otherwise we could make, for instance, the *overusing* function from Section 3.1.3. You can check that $1 \cdot q; f{:}_\omega a \to_\omega b, x{:}_\omega 1 \cdot q \multimap a \vdash f\ x : b$ indeed does not type check, because the scaling of Q_2 in rule E-App ensures that the constraint would be $\omega \cdot q$ instead. On the other hand, it is perfectly fine to have $1 \cdot q; f{:}_\omega a \to_1 b, x{:}_\omega 1 \cdot q \multimap a \vdash f\ x : b$ when f is a linear function.

Variables. As is standard, the rule E-Var rule works in a context containing more than just the used binding for x. However, crucially, our rule allows only *unrestricted* variables to be discarded; linear variables *must* be used. We can see this in the rule by noticing that the context has an

unrestricted component $\omega \cdot \Gamma_2$. The Γ_1 component might be restricted or might not, allowing this rule to apply both for restricted and unrestricted x.

Data constructors. Data constructors K don't have a dedicated typing rule. Instead they are typed using the rule E-Var, where they are treated as if they were unrestricted variables.

Let-bindings. Bindings in a **let** may be for either linear or unrestricted variables. We could require all bindings to be linear and to implement unrestricted information only using *Ur*, but it is very easy to add a multiplicity annotation on **let**, and so we do.

Local assumptions. Rule E-Let includes support for local assumptions. We thus have the ability to generalise a subset of the constraints needed by e_1 (but not the type variables—no **let**-generalisation here, though it could be added). The inference algorithm of Section 6 will not make use of this possibility.

Existentials. We include $\exists \overline{a}.\tau \circledcirc Q$, as introduced in Section 3, together with the $\Box$ constructor. See rules E-Pack and E-Unpack.

6 CONSTRAINT INFERENCE

The type system of Figure 6 gives a declarative description of what programs are acceptable. We now present the algorithmic counterpart to this system. Our algorithm is structured, unsurprisingly, around generating and solving constraints, broadly following the template of Pottier and Rémy [2005]. That is, our algorithm takes a pass over the abstract syntax entered by the user, generating constraints as it goes. Then, separately, we solve those constraints (that is, try to satisfy them) in the presence of a set of assumptions, or we determine that the assumptions do not imply that the constraints hold. In the latter case, we issue an error to the programmer.

The procedure is responsible for inferring both *types* and *constraints*. For our type system, type inference can be done independently from constraint inference. Indeed, we focus on the latter, and defer type inference to an external oracle (such as [Matsuda 2020]). That is, we assume an algorithm that produces typing derivations for the judgement $\Gamma \vdash e : \tau$, ignoring all the constraints. Then, we describe a constraint generation algorithm that passes over these typing derivations. We can make this simplification for two reasons:

- We do not formalise type equality constraints, and our implementation in GHC (Section 8.2.2) takes care to not allow linear equality constraints to influence type inference. Indeed, a typical treatment of unification would be unsound for linear equalities, because it reuses the same equality many times (or none at all). Linear equalities make sense (Shulman [2018] puts linear equalities to great use), but they do not seem to lend themselves to automation.
- We do not support, or intend to support, multiplicity polymorphism in constraint arrows. That is, the multiplicity of a constraint is always syntactically known to be either linear or unrestricted. This way, no equality constraints (which might, conceivably, relate multiplicity variables) can interfere with constraint resolution.

6.1 Wanted Constraints

The constraints C generated in our system have a richer logical structure than the simple constraints Q, above. Following GHC and echoing Vytiniotis et al. [2011], we call these *wanted constraints*: they are constraints which the constraint solver *wants* to prove. An unproved wanted constraint results in a type error reported to the programmer.

$$C \quad ::= \quad Q \mid C_1 \otimes C_2 \mid C_1 \,\&\, C_2 \mid \pi \cdot (Q \multimap C) \qquad \text{Wanted constraints}$$

$\boxed{Q \vdash C}$ *(Wanted-constraint entailment)*

$$\text{C-Dom}\ \frac{Q_1 \Vdash Q_2 \quad Q_2 \vdash Q_3}{Q_1 \vdash Q_3} \qquad \text{C-Id}\ \frac{}{Q \vdash Q} \qquad \text{C-Tensor}\ \frac{Q_1 \vdash C_1 \quad Q_2 \vdash C_2}{Q_1 \otimes Q_2 \vdash C_1 \otimes C_2} \qquad \text{C-With}\ \frac{Q \vdash C_1 \quad Q \vdash C_2}{Q \vdash C_1 \,\&\, C_2} \qquad \text{C-Impl}\ \frac{Q_0 \otimes Q_1 \vdash C}{\pi \cdot Q_0 \vdash \pi \cdot (Q_1 \multimap C)}$$

Fig. 7. Wanted-constraint entailment

A simple constraint is a valid wanted constraint, and we have two forms of conjunction for wanted constraints: the new $C_1 \,\&\, C_2$ construction (read C_1 *with* C_2), alongside the more typical $C_1 \otimes C_2$. These are connectives from linear logic: $C_1 \otimes C_2$ is the *multiplicative* conjunction, and $C_1 \,\&\, C_2$ is the *additive* conjunction. Both connectives are conjunctions, but they differ in meaning. To satisfy $C_1 \otimes C_2$ one consumes the (linear) assumptions consumed by satisfying C_1 and those consumed by C_2; if an assumed linear constraint is needed to prove both C_1 and C_2, then $C_1 \otimes C_2$ will not be provable, because that linear assumption cannot be used twice. On the other hand, satisfying $C_1 \,\&\, C_2$ requires that satisfying C_1 and C_2 must each consume the *same* assumptions, which $C_1 \,\&\, C_2$ consumes as well. Thus, if C is assumed linearly (and we have no other assumptions), then $C \otimes C$ is not provable, while $C \,\&\, C$ is. The intuition, here, is that in $C_1 \,\&\, C_2$, only one of C_1 or C_2 will be eventually used. "With" constraints arise from the branches in a **case**-expression.

The last form of wanted constraint C is an implication $\pi \cdot (Q \multimap C)$. The more interesting case is $\omega \cdot (Q \multimap C)$: to prove $\omega \cdot (Q \multimap C)$, you need to prove C under the *linear* assumption Q, but without using any other linear assumptions.

These implications arise when we unpack an existential package that contains a linear constraint and also when checking a **let**-binding. We can define scaling over wanted constraints by recursion as follows, where we use scaling over simple constraints in the simple-constraint case:

$$\begin{cases} \pi \cdot (C_1 \otimes C_2) & = & \pi \cdot C_1 \otimes \pi \cdot C_2 \\ 1 \cdot (C_1 \,\&\, C_2) & = & C_1 \,\&\, C_2 \\ \omega \cdot (C_1 \,\&\, C_2) & = & \omega \cdot C_1 \otimes \omega \cdot C_2 \\ \pi \cdot (\rho \cdot (Q \multimap C)) & = & (\pi \cdot \rho) \cdot (Q \multimap C) \end{cases}$$

For the most part, scaling of wanted constraints is straightforward. The only peculiar case is when we scale the additive conjunction $C_1 \,\&\, C_2$ by ω, the result is a multiplicative conjunction. The intuition here is that when if we have both $\omega \cdot C_1$ and $\omega \cdot C_2$, then a choice between C_1 and C_2 can be made ω times.

We define an entailment relation over wanteds in Figure 7. Note that this relation uses only simple constraints Q as assumptions, as there is no way to assume the more elaborate C[6].

Before we move on to constraint generation proper, let us highlight a few technical, yet essential, lemmas about the wanted-constraint entailment relation.

Lemma 6.1 (Inversion). *The inference rules of $Q \vdash C$ can be read bottom-up (up to the set $\mathcal{D}$) as well as top-down, as is required of $Q_1 \Vdash Q_2$ in Figure 4. That is:*

- *If $Q \vdash C_1 \otimes C_2$, then there exists Q_1, $Q_{\mathcal{D}}$ and Q_2 such that $Q_{\mathcal{D}} \in \mathcal{D}$, $Q_1 \otimes Q_{\mathcal{D}} \vdash C_1$, $Q_{\mathcal{D}} \otimes Q_2 \vdash C_2$, and $Q = Q_1 \otimes Q_{\mathcal{D}} \otimes Q_2$.*
- *If $Q \vdash C_1 \,\&\, C_2$, then $Q \vdash C_1$ and $Q \vdash C_2$.*
- *If $Q \vdash \pi \cdot (Q_2 \multimap C)$, then there exists Q_1 such that $Q_1 \otimes Q_2 \vdash C$ and $Q = \pi \cdot Q_1$*

[6] Allowing the full wanted-constraint syntax in assumptions is the subject of work by Bottu et al. [2017].

$\boxed{\Gamma \mapsto e : \tau \rightsquigarrow C}$ *(Constraint generation)*

$$\textsc{G-Var}\ \frac{\Gamma_1 = x{:}_1 \forall \overline{a}.Q \multimap v}{\Gamma_1 + \omega \cdot \Gamma_2 \mapsto x : v[\overline{\tau}/\overline{a}] \rightsquigarrow Q[\overline{\tau}/\overline{a}]} \qquad \textsc{G-Abs}\ \frac{\Gamma, x{:}_\pi \tau_0 \mapsto e : \tau \rightsquigarrow C}{\Gamma \mapsto \lambda x.e : \tau_0 \to_\pi \tau \rightsquigarrow C}$$

$$\textsc{G-App}\ \frac{\Gamma_1 \mapsto e_1 : \tau_2 \to_\pi \tau \rightsquigarrow C_1 \quad \Gamma_2 \mapsto e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma_1 + \pi \cdot \Gamma_2 \mapsto e_1\ e_2 : \tau \rightsquigarrow C_1 \otimes \pi \cdot C_2}$$

$$\textsc{G-Pack}\ \frac{\Gamma \mapsto e : \tau[\overline{v}/\overline{a}] \rightsquigarrow C}{\Gamma \mapsto \Box e : \exists \overline{a}.\tau \otimes Q \rightsquigarrow C \otimes Q[\overline{v}/\overline{a}]}$$

$$\textsc{G-Unpack}\ \frac{\Gamma_1 \mapsto e_1 : \exists \overline{a}.\tau_1 \otimes Q_1 \rightsquigarrow C_1 \quad \overline{a}\ \text{fresh} \quad \Gamma_2, x{:}_1 \tau_1 \mapsto e_2 : \tau \rightsquigarrow C_2}{\Gamma_1 + \Gamma_2 \mapsto \textbf{let}\ \Box x = e_1\ \textbf{in}\ e_2 : \tau \rightsquigarrow C_1 \otimes 1 \cdot (Q_1 \multimap C_2)}$$

$$\textsc{G-Case}\ \frac{\Gamma \mapsto e : T\,\overline{\sigma} \rightsquigarrow C \quad K_i : \forall \overline{a}.\overline{v_i} \to_{\overline{\pi_i}} T\,\overline{a} \quad \Delta, x_i{:}_{(\pi \cdot \pi_i)} v_i[\overline{\sigma}/\overline{a}] \mapsto e_i : \tau \rightsquigarrow C_i}{\pi \cdot \Gamma + \Delta \mapsto \textbf{case}_\pi\ e\ \textbf{of}\ \{\overline{K_i\ \overline{x_i} \to e_i}\} : \tau \rightsquigarrow \pi \cdot C \otimes \&\, C_i}$$

$$\textsc{G-Let}\ \frac{\Gamma_1 \mapsto e_1 : \tau_1 \rightsquigarrow C_1 \quad \Gamma_2, x{:}_\pi \tau_1 \mapsto e_2 : \tau \rightsquigarrow C_2}{\pi \cdot \Gamma_1 + \Gamma_2 \mapsto \textbf{let}_\pi\ x = e_1\ \textbf{in}\ e_2 : \tau \rightsquigarrow \pi \cdot C_1 \otimes C_2}$$

$$\textsc{G-LetSig}\ \frac{\Gamma_1 \mapsto e_1 : \tau_1 \rightsquigarrow C_1 \quad \overline{a}\ \text{fresh} \quad \Gamma_2, x{:}_\pi \forall \overline{a}.Q \multimap \tau_1 \mapsto e_2 : \tau \rightsquigarrow C_2}{\pi \cdot \Gamma_1 + \Gamma_2 \mapsto \textbf{let}_\pi\ x : \forall \overline{a}.Q \multimap \tau_1 = e_1\ \textbf{in}\ e_2 : \tau \rightsquigarrow C_2 \otimes \pi \cdot (Q \multimap C_1)}$$

Fig. 8. Constraint generation

Lemma 6.2 (Scaling). *If $Q \vdash C$, then $\pi \cdot Q \vdash \pi \cdot C$.*

Lemma 6.3 (Inversion of scaling). *If $Q \vdash \pi \cdot C$ then $Q' \vdash C$ and $Q = \pi \cdot Q'$ for some Q'.*

6.2 Constraint Generation

The process of inferring constraints is split into two parts: generating constraints, which we do in this section, then solving them in Section 6.3. Constraint generation is described by the judgement $\Gamma \mapsto e : \tau \rightsquigarrow C$ (defined in Figure 8) which outputs a constraint C required to make e typecheck. The definition $\Gamma \mapsto e : \tau \rightsquigarrow C$ is syntax directed, so it can directly be read as an algorithm, taking as input a *typing derivation* for $\Gamma \vdash e : \tau$ (produced by an external type inference oracle as discussed above). Notably, the algorithm has access to the context splitting from the (previously computed) typing derivation, and is thus indeed syntax directed.

The rules of Figure 8 constitute a mostly unsurprising translation of the rules of Figure 6, except for the following points of interest:

Case expressions. Note the use of $\&$ in the conclusion of rule G-Case. We require that each branch of a **case** expression use the exact same (linear) assumptions; this is enforced by combining the emitted constraints with $\&$, not $\otimes$. This can also be understood in terms of the array example of Section 1: if an array is freed in one branch of a **case**, we require it to be freed (or freezed) in the other branches too. Otherwise, the array's state will be unknown to the type system after the **case**.

Implications. The introduction of constraints local to a definition (rule G-LetSig) corresponds to emitting an implication constraint.

Unannotated **let**. However, the G-Let rule does not produce an implication constraint, as we do not model **let**-generalisation [Vytiniotis et al. 2010].

The key property of the constraint-generation algorithm is that, if the generated constraint is solvable, then we can indeed type the term in the qualified type system of Section 5. That is, these rules are simply an implementation of our declarative qualified type system.

Lemma 6.4 (Soundness of constraint generation). *For all Q_g, if $\Gamma \mapsto e : \tau \rightsquigarrow C$ and $Q_g \vdash C$ then $Q_g; \Gamma \vdash e : \tau$.*

6.3 Constraint Solving

In this section, we build a *constraint solver* that proves that $Q_g \vdash C$ holds, as required by Lemma 6.4. The constraint solver is represented by the following judgement:

$$U; D; L_i \vdash_{\mathsf{s}} C \rightsquigarrow L_o$$

The judgement takes in three contexts: U, which holds all the unrestricted atomic constraint assumptions, D which holds the linear atomic assumptions which are members of $\mathcal{D}$ and L_i, which holds the linear atomic constraint assumptions which aren't members of $\mathcal{D}$. The linear contexts D, L_i, and L_o have been described as multisets (Section 5.1), but we treat them as ordered lists in the more concrete setting here; we will see soon why this treatment is necessary.

Linearity requires treating constraints as consumable resources. This is what L_o is for: it contains the hypotheses of L_i which are not consumed when proving C. As suggested by the notation, it is an output of the algorithm. Constraints from D are never outputted in L_i: if constraints from D remain unused, we weaken them instead.

If the constraint solver finds a solution, then the output linear constraints must be a subset of the input linear constraints, and the solution must indeed be entailed from the given assumptions.

Lemma 6.5 (Constraint solver soundness). *If $U; D; L_i \vdash_{\mathsf{s}} C \rightsquigarrow L_o$, then:*

(1) $L_o \subseteq L_i$
(2) $(U, D \uplus L_i) \vdash C \otimes (\emptyset, L_o)$

To handle simple wanted constraints, we will need a domain-specific *atomic-constraint solver* to be the algorithmic counterpart of the abstract entailment relation of Section 5.1. The main solver will appeal to this atomic-constraint solver when solving atomic constraints. The atomic-constraint solver is represented by the following judgement:

$$U; D; L_i \vdash^{\mathsf{a}}_{\mathsf{s}} \pi \cdot q \rightsquigarrow L_o$$

It has a similar structure to the main solver, but only deals with atomic constraints. Even though the main solver is parameterised by this atomic-constraint solver, we will give an instantiation in Section 6.3.2. We require the following property of the atomic-constraint solver:

Property 6.6 (Atomic-constraint solver soundness). *If $U; D; L_i \vdash^{\mathsf{a}}_{\mathsf{s}} \pi \cdot q \rightsquigarrow L_o$, then:*

(1) $L_o \subseteq L_i$
(2) $(U, D \uplus L_i) \Vdash \pi \cdot q \otimes (\emptyset, L_o)$

6.3.1 Constraint Solver Algorithm. Building on this atomic-constraint solver, we use a linear proof search algorithm based on the recipe given by Cervesato et al. [2000]. Figure 9 presents the rules of the constraint solver.

- The S-Mult rule proceeds by solving one side of a conjunction first, then passing the output constraints to the other side. Both the unrestricted context and the duplicable context are shared between both sides.
- The S-Add rule handles additive conjunction. The linear constraints are shared between the branches (since additive conjunction is generated from **case** expressions, only one of them is actually going to be executed). Both branches must consume exactly the same resources.

$\boxed{U; D; L_i \vdash_{\mathsf{s}} C_w \rightsquigarrow L_o}$ *(Constraint solving)*

$$\textsc{S-Atom}\ \frac{U; D; L_i \vdash^{\mathsf{a}}_{\mathsf{s}} \pi \cdot q \rightsquigarrow L_o}{U; D; L_i \vdash_{\mathsf{s}} \pi \cdot q \rightsquigarrow L_o} \qquad \textsc{S-Mult}\ \frac{U; D; L_i \vdash_{\mathsf{s}} C_1 \rightsquigarrow L'_o \quad U; D; L'_o \vdash_{\mathsf{s}} C_2 \rightsquigarrow L_o}{U; D; L_i \vdash_{\mathsf{s}} C_1 \otimes C_2 \rightsquigarrow L_o}$$

$$\textsc{S-ImplOne}\ \frac{U \cup U_0; D \uplus (L_0 \cap \mathcal{D}); L_i \uplus (L_0 \setminus \mathcal{D}) \vdash_{\mathsf{s}} C \rightsquigarrow L_o \quad L_o \subseteq L_i}{U; D; L_i \vdash_{\mathsf{s}} 1 \cdot ((U_0, L_0) \multimap C) \rightsquigarrow L_o} \qquad \textsc{S-Add}\ \frac{U; D; L_i \vdash_{\mathsf{s}} C_1 \rightsquigarrow L_o \quad U; D; L_i \vdash_{\mathsf{s}} C_2 \rightsquigarrow L_o}{U; D; L_i \vdash_{\mathsf{s}} C_1 \,\&\, C_2 \rightsquigarrow L_o}$$

$$\textsc{S-ImplMany}\ \frac{U \cup U_0; L_0 \cap \mathcal{D}; L_0 \setminus \mathcal{D} \vdash_{\mathsf{s}} C \rightsquigarrow \emptyset}{U; D; L_i \vdash_{\mathsf{s}} \omega \cdot ((U_0, L_0) \multimap C) \rightsquigarrow L_i}$$

Fig. 9. Constraint solver

- The S-ImplOne rules handles linear implications. The unrestricted and linear components of the assumption are unioned with their respective context when solving the conclusion. Note how the linear constraints, in particular, are classified according to whether they are members of $\mathcal{D}$ or not. Importantly (see Section 6.3.2), the linear assumptions are added to the front of the lists. The side condition that the output context is a subset of the input context ensures that the implication fully consumes its assumption and does not leak it to the ambient context.
- The S-ImplMany rules handles unrestricted implication. The conclusion uses its own linear assumption, but none of the other linear constraints. This is because, as per C-Impl, unrestricted implications can only use an unrestricted context. In particular, crucially, the constraints from D, despite being duplicable and discardable, are not (and cannot) be used to prove an unrestricted implication, as first discussed in Section 5.1.

6.3.2 An Atomic-Constraint Solver. So far, the atomic-constraint domain has been an abstract parameter. In this section, though, we offer a concrete domain which supports our examples.

For the sake of our examples, we need very little: linear constraints can remain abstract. It is thus sufficient for the entailment relation (Figure 10a) to prove q if and only if it is already assumed—while respecting linearity. That is, with the exception of a distinguished constraint $\mathcal{L}$, which can be duplicated and discarded, and we use to model the *Linearly* constraint from 3.2. The set $\mathcal{D}$ is defined to only contain $\mathcal{L}$, therefore the D context is a sequence of 0 or more $\mathcal{L}$.

The corresponding atomic-constraint solver (Figure 10b) is more interesting. It is deterministic: in all circumstances, only one of the three rules can apply. This means that the algorithm does not guess, thus never needs to backtrack. Avoiding guesses is a key property of GHC's solver [Vytiniotis et al. 2011, Section 6.4], one we must maintain if we are to be compatible with GHC.

Figure 10b is also where the fact that the L are lists comes into play. Indeed, rule Atom-OneL takes care to use the most recent occurrence of q (remember that rule S-ImplOne adds the new hypotheses on the front of the list). To understand why, consider the following example:

$f = \mathit{linearly}\ \$$
  let $\square(\mathit{Ur}\ \mathit{arr}) = \mathit{new}\ 10$
    $\mathit{fr} :: \mathit{RW}\ n \multimap ()$

$\boxed{Q_1 \Vdash Q_2}$ *(Entailment relation)*

$$\text{Q-Hyp}\ \frac{}{\omega\cdot Q \otimes \pi\cdot q \Vdash \pi\cdot q} \qquad \text{Q-Prod}\ \frac{Q_1 \Vdash Q_1' \quad Q_2 \Vdash Q_2'}{Q_1 \otimes Q_2 \Vdash Q_1' \otimes Q_2'} \qquad \text{Q-Empty}\ \frac{}{\omega\cdot Q \Vdash \varepsilon} \qquad \text{Q-DiscardD}\ \frac{}{1\cdot \mathcal{L} \Vdash \varepsilon} \qquad \text{Q-DupD}\ \frac{}{1\cdot \mathcal{L} \Vdash 1\cdot\mathcal{L} \otimes 1\cdot\mathcal{L}}$$

(a) Entailment relation

$\boxed{U; D; L \vdash^{\mathsf{a}}_{\mathsf{s}} \pi\cdot q \rightsquigarrow L_o}$ *(Atomic-constraint solver)*

$$\text{Atom-Many}\ \frac{q \in U}{U; D; L \vdash^{\mathsf{a}}_{\mathsf{s}} \omega\cdot q \rightsquigarrow L} \qquad \text{Atom-OneL}\ \frac{L = L_1, q, L_2 \quad q \notin L_2 \quad q \notin U}{U; D; L \vdash^{\mathsf{a}}_{\mathsf{s}} 1\cdot q \rightsquigarrow L_1, L_2} \qquad \text{Atom-OneD}\ \frac{\mathcal{L} \notin U}{U; D, \mathcal{L}; L \vdash^{\mathsf{a}}_{\mathsf{s}} 1\cdot q \rightsquigarrow L} \qquad \text{Atom-OneU}\ \frac{q \in U \quad q \notin D \uplus L}{U; D; L \vdash^{\mathsf{a}}_{\mathsf{s}} 1\cdot q \rightsquigarrow L}$$

(b) Atomic-constraint solver

Fig. 10. A stripped-down constraint domain

$$\begin{aligned} &fr = free\ arr \\ &() = fr \qquad\qquad \textbf{in}\ Ur\ () \end{aligned}$$

In this example, the programmer meant for *free* to use the $RW\ n$ constraint introduced locally in the type of *fr*. Yet there are actually two linear $RW\ n$ constraints: this local one and the one assumed when unpacking *arr*. The wrong choice among the constraints will lead the algorithm to fail. Choosing the first q linear assumption guarantees we get the most local one.

Another interesting feature of the solver (Figure 10b) is that no rule solves a linear constraint if it appears both in the unrestricted and a linear context. Consider the following (contrived) API:

class C

$giveC :: (C \Rightarrow Int) \to Int$ $\qquad$ $useC :: C \multimap Int$

giveC gives an unrestricted copy of C to some continuation, while *useC* uses C linearly. Now consider two potential consumers of this API:

$ambiguous1 :: C \multimap Int$ $\qquad$ $ambiguous2 :: C \multimap (Int, Int)$

$ambiguous1 = giveC\ useC$ $\qquad$ $ambiguous2 = (giveC\ useC, useC)$

Looking at *ambiguous1*, the invocation of *useC* has both a linear C in scope, and a more local unrestricted C. The strategy to pick the more local constraint fails here, because it would leave the linear C unconsumed. A tempting refinement might be to always consume the most local *linear* constraint. That would handle handles *ambiguous1* correctly, but fail on *ambiguous2*. In the case of the latter, if the invocation of *giveC useC* consumes the linear C, then the second *useC* invocation will fail. It is possible to give a type derivation to *ambiguous2* in the qualified type system of Section 5 by making the first *useC* consume the unrestricted C and the second *useC* consume the linear C. This assignment, however, would require the constraint solver to guess when solving the constraint from the first *useC*. Accordingly, in order to both avoid backtracking and to keep type inference independent of the order terms appear in the program text, *bad* is rejected. This introduces incompleteness with respect the entailment relation. We conjecture that this is the only source of incompleteness that we introduce beyond what is already in GHC [Vytiniotis et al. 2011, Section 6].

$$
\begin{array}{llll}
\sigma & ::= & \forall \overline{a}.\tau & \text{Type schemes} \\
\tau, \upsilon & ::= & \ldots \mid \exists \overline{a}.\tau \otimes \upsilon & \text{Types} \\
e & ::= & \ldots \mid \Box\,(e_1, e_2) \mid \textbf{let}\ \Box(x, y) = e_1\ \textbf{in}\ e_2 & \text{Expressions}
\end{array}
$$

$\boxed{\Gamma \vdash e : \tau}$ *(Core language typing)*

$$
\textsc{L-Pack}\ \frac{\Gamma_1 \vdash e_1 : \tau_1[\overline{\upsilon}/\overline{a}] \qquad \Gamma_2 \vdash e_2 : \tau_2[\overline{\upsilon}/\overline{a}]}{\Gamma_1 + \Gamma_2 \vdash \Box\,(e_1, e_2) : \exists \overline{a}.\tau_2 \otimes \tau_1}
\qquad
\textsc{L-Unpack}\ \frac{\Gamma_1 \vdash e_1 : \exists \overline{a}.\tau_2 \otimes \tau_1 \qquad \overline{a}\ \text{fresh} \qquad \Gamma_2, x{:}_1\tau_1, y{:}_1\tau_2 \vdash e_2 : \tau}{\Gamma_1 + \Gamma_2 \vdash \textbf{let}\ \Box(x, y) = e_1\ \textbf{in}\ e_2 : \tau}
$$

Fig. 11. Core calculus (subset)

7 DESUGARING

The semantics of our language is given by desugaring it into a simpler core language: a variant of the λ^q calculus [Bernardy et al. 2017]. We define the core language's type system here; its operational semantics is the same, *mutatis mutandis*, as that of Linear Haskell.

7.1 The Core Calculus

The core calculus is a variant of the type system defined in Section 5, but without constraints. That is, the evidence for constraints is passed explicitly in this core calculus. Following λ^q, we assume the existence of the following data types:

- $\tau_1 \otimes \tau_2$ with sole constructor $(,) : \forall a\, b.a \to_1 b \to_1 a \otimes b$. We will write (e_1, e_2) for $(,)\ e_1\ e_2$.
- $\mathbf{1}$ with sole constructor $() : \mathbf{1}$.
- $\mathbf{Ur}\ \tau$ with sole constructor $\mathbf{Ur} : \forall a.a \to_\omega \mathbf{Ur}\ a$

Figure 11 highlights the differences from the qualified system:

- Type schemes σ do not support qualified types.
- Existentially quantified types ($\exists \overline{a}.\tau \otimes Q$) are now represented as an (existentially quantified, linear) pair of values ($\exists \overline{a}.\tau_2 \otimes \tau_1$). Accordingly, $\Box$ operates on pairs.

The differences between our core calculus and λ^q are as follows:

- We do not support multiplicity polymorphism.
- On the other hand, we do include type polymorphism.
- Polymorphism is implicit rather than explicit. This is not an essential difference, but it simplifies the presentation. We could, for example, include more details in the terms in order to make type-checking more obvious; this amounts essentially to an encoding of typing derivations in the terms[7].
- We have existential types. These can be realised in regular Haskell as a family of datatypes.

Using Lemma 6.4 together with Lemma 6.5 we know that if $\Gamma \mapsto e : \tau \rightsquigarrow C$ and $U; D; L \vdash_{\mathsf{s}} C \rightsquigarrow \emptyset$, then $(U, D \uplus L); \Gamma \vdash e : \tau$. It only remains to desugar derivations of $Q; \Gamma \vdash e : \tau$ into the core calculus.

7.2 From Qualified to Core

7.2.1 Evidence. In order to desugar derivations of the qualified system to the core calculus, we pass evidence explicitly[8]. To do so, we require some more material from constraints. Namely, we

[7] See, for example, Weirich et al. [2017] and their comparison between an implicit core language D and an explicit one DC.
[8] This technique is also often called dictionary-passing style [Hall et al. 1996] because, in the case of type classes, evidences are dictionaries, and because type classes were the original form of constraints in Haskell.

$$\begin{cases} [\![1 \cdot q]\!]^{\mathbf{ev}} & = & [\![q]\!]^{\mathbf{ev}} \\ [\![\omega \cdot q]\!]^{\mathbf{ev}} & = & \mathbf{Ur}\,([\![q]\!]^{\mathbf{ev}}) \\ [\![\varepsilon]\!]^{\mathbf{ev}} & = & \mathbf{1} \\ [\![Q_1 \otimes Q_2]\!]^{\mathbf{ev}} & = & [\![Q_1]\!]^{\mathbf{ev}} \otimes [\![Q_2]\!]^{\mathbf{ev}} \end{cases}$$

(a) Evidence passing

$$\begin{cases} [\![Q; \Gamma \vdash x : v[\overline{\tau}/\overline{a}]]\!]_z = x\ z \\ [\![Q \otimes Q_1[\overline{v}/\overline{a}]; \Gamma \vdash \Box e : \exists\ \overline{a}.\tau \circledcirc Q_1]\!]_z = \\ \quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z', z'') \to \\ \quad\quad \Box(z'', [\![Q; \Gamma \vdash e : \tau[\overline{v}/\overline{a}]]\!]_{z'})\} \\ [\![Q_1 \otimes Q_2; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}\ \Box x = e_1\ \mathbf{in}\ e_2 : \tau]\!]_z = \\ \quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, z_2) \to \\ \quad\quad \mathbf{let}\ \Box z', x = [\![Q_1; \Gamma_1 \vdash e_1 : \exists\ \overline{a}.\tau_1 \circledcirc Q]\!]_{z_1}\ \mathbf{in} \\ \quad\quad \mathbf{let}_1\ z_2' = (z_2, z')\ \mathbf{in} \\ \quad\quad [\![Q_2 \otimes Q; \Gamma_2, x :_1 \tau_1 \vdash e_2 : \tau]\!]_{z_2'}\} \\ [\![Q; \Gamma \vdash e : \tau]\!]_z = \quad \text{-- rule E-Sub} \\ \quad \mathbf{let}_1\ z' = [\![Q \Vdash Q_1]\!]^{\mathbf{ev}}\ z\ \mathbf{in}\ [\![Q_1; \Gamma \vdash e : \tau]\!]_{z'} \\ \quad \ldots \end{cases}$$

(b) Desugaring (subset)

Fig. 12. Evidence passing and desugaring

assume a type $[\![q]\!]^{\mathbf{ev}}$ for each atomic constraint q, defined in Figure 12a. The $[\![\_]\!]^{\mathbf{ev}}$ operation extends to simple constraints as $[\![Q]\!]^{\mathbf{ev}}$. Furthermore, we require that for every Q_1 and Q_2 such that $Q_1 \Vdash Q_2$, there is a (linear) function $[\![Q_1 \Vdash Q_2]\!]^{\mathbf{ev}} : [\![Q_1]\!]^{\mathbf{ev}} \to_1 [\![Q_2]\!]^{\mathbf{ev}}$.

Let us now define a family of functions $[\![\_]\!]$ to translate the type schemes, types, contexts, and typing derivations of the qualified system into the types, type schemes, contexts, and terms of the core calculus.

7.2.2 Translating Types. Type schemes σ are translated by turning the implicit argument Q into an explicit one of type $[\![Q]\!]^{\mathbf{ev}}$. Translating types τ and contexts Γ proceeds as expected.

$$\begin{cases} [\![\forall \overline{a}.Q \Rightarrow \tau]\!] & = & \forall \overline{a}.[\![Q]\!]^{\mathbf{ev}} \to_1 [\![\tau]\!] \\ \begin{cases} [\![\tau_1 \to_\pi \tau_2]\!] & = & [\![\tau_1]\!] \to_\pi [\![\tau_2]\!] \\ [\![\exists \overline{a}.\tau \circledcirc Q]\!] & = & \exists \overline{a}.[\![\tau]\!] \otimes [\![Q]\!]^{\mathbf{ev}} \end{cases} \end{cases} \qquad \begin{cases} [\![\bullet]\!] & = & \bullet \\ [\![\Gamma, x :_\pi \tau]\!] & = & [\![\Gamma]\!], x :_\pi [\![\tau]\!] \end{cases}$$

7.2.3 Translating Terms. Given a derivation $Q; \Gamma \vdash e : \tau$, we can build an expression $[\![Q; \Gamma \vdash e : \tau]\!]_z$, such that $[\![\Gamma]\!], z :_1 [\![Q]\!]^{\mathbf{ev}} \vdash [\![Q; \Gamma \vdash e : \tau]\!]_z : [\![\tau]\!]$ (for some fresh variable z). Even though we abbreviate the derivation as only its concluding judgement, the translation is defined recursively on the whole typing derivation: in particular, we have access to typing rule premises in the body of the definition. We present some of the interesting cases in Figure 12b.

The cases correspond to the E-Var, E-Unpack[9], and E-Sub rules, respectively. Variables are stored with qualified types in the environment, so they get translated to functions that take the evidence as argument. Accordingly, the evidence is inserted by passing z as an argument. Handling E-Unpack requires splitting the context into two: e_1 is desugared as a pair, and the evidence it contains is passed to e_2. Finally, subsumption summons the function corresponding to the entailment relation $Q \Vdash Q_1$ and applies it to $z : [\![Q]\!]^{\mathbf{ev}}$ then proceeds to desugar e with the resulting evidence

[9]The attentive reader may note that the case for **let**□ extracts out Q_1 and Q_2 from the provided simple constraint. Given that simple constraints Q have no internal ordering and allow duplicates (in the non-linear component), this splitting is not well defined. To fix this, an implementation would have to *name* individual components of Q, and then the typing derivation can indicate which constraints go with which sub-expression. Happily, GHC *already* names its constraints, and so this approach fits easily in the implementation. We could also augment our formalism here with these details, but they add clutter with little insight.

for Q_1. Crucially, since $[\![\_]\!]_z$ is defined on *derivations*, we can access the premises used in the rule. Namely, Q_1 is available in this last case from the E-Sub rule's premise.

It is straightforward by induction, to verify that desugaring is correct:

Theorem 7.1 (Desugaring). *If* $Q; \Gamma \vdash e : \tau$, *then* $[\![\Gamma]\!], z{:}_1 [\![Q]\!]^{\mathbf{ev}} \vdash [\![Q; \Gamma \vdash e : \tau]\!]_z : [\![\tau]\!]$, *for any fresh variable* z.

Thanks to the desugaring machinery, the semantics of a language with linear constraints can be understood in terms of a simple core language with linear types, such as λ^q, or indeed, GHC Core.

8 INTEGRATING INTO GHC

One of the guiding principles behind our design was ease of integration with modern Haskell. In this section we describe some of the particulars of adding linear constraints to GHC.

8.1 Implementation

We have written a prototype implementation [Kiss et al. 2022] of linear constraints on top of GHC 9.1, a version that already ships with the `LinearTypes` extension. Function arrows ($\to$) and context arrows ($\Rightarrow$) share the same internal representation in the typechecker, differentiated only by a boolean flag. Thus, the `LinearTypes` implementation effort has already laid down the bureaucratic ground work of annotating these arrows with multiplicity information.

The key changes affect constraint generation and constraint solving. Constraints are now annotated with a multiplicity, according to the context from which they arise. With `LinearTypes`, GHC already scales the usage of term variables. We simply modified the scaling function to capture all the generated constraints and re-emit a scaled version of them, which is a fairly local change.

The constraint solver maintains a set of given constraints (the *inert set* in GHC jargon), which corresponds to the U, D, and L contexts in our solver judgements in Section 6.3. When the solver goes under an implication, the assumptions of the implication are added to set of givens. When a new given is added, we record the *level* of the implication (how many implications deep the constraint arises from) along with the constraint. So that in case there are multiple matching givens, the constraint solver selects the innermost one (in Section 6.3 we use an ordered list for this purpose).

As constraint solving proceeds, the compiler pipeline constructs a term in a typed language known as GHC Core [Sulzmann et al. 2007]. In Core, type class constraints are turned into explicit evidence (see Section 7). Thanks to being fully annotated, Core has decidable typechecking, which is used to find and fix bugs in the compiler (the Haskell type checker finds mistakes in user programs). Thus, the Core typechecker verifies that the desugaring procedure produced a linearity-respecting program before code generation occurs.

8.2 Interaction with Other Features

Since constraints play an important role in GHC's type system, we must pay close attention to the interaction of linearity with other language features related to constraints. Of these, we point out two that require some extra care.

8.2.1 Superclasses. Haskell's type classes can have *superclasses*, which place constraints on all of the instances of that class. For example, the *Ord* class is defined as

class $Eq\ a \Rightarrow Ord\ a$ **where** ...

which means that every ordered type must also support equality. Such superclass declarations extend the entailment relation: if we know that a type is ordered, we also know that it supports equality. This is troublesome if we have a linear occurrence of $Ord\ a$, because then using this

entailment, we could conclude that a linear constraint (*Ord a*) implies an unrestricted constraint (*Eq a*), which violates Lemma 5.5.

But even linear superclass constraints cause trouble. Consider a version of *Ord a* that has *Eq a* as a linear superclass.

class *Eq a* ⊸ *Ord a* **where** ...

When given a linear *Ord a*, should we keep it as *Ord a*, or rewrite to *Eq a* using the entailment? Short of backtracking, the constraint solver needs to make a guess, which GHC never does.

To address both of these issues at once, we make the following rule: the superclasses of a linear constraint are ignored.

8.2.2 Equality Constraints. In Section 6 we argued that *type* inference and *constraint* inference can be performed independently. However, this is not the case for GHC's constraint domain, because it supports equality constraints, which allows unification problems to be deferred, and potentially be solvable only after solving other constraints first.

To reconcile this with our presentation, we need to ensure that *unrestricted constraint* inference and *linear constraint* inference can be performed independently. That is, solving a linear constraint should never be required for solving an unrestricted constraint. This is ensured by Lemma 5.5.

They key is to represent unification problems as *unrestricted* equality constraints, so a given linear equality constraint cannot be used during type inference. This way, linear equalities require no special treatment, and are harmless.

8.3 Inferring Packing and Unpacking

Recent work [Eisenberg et al. 2021] describes an algorithm (call it EDWL, after the authors' names) that can infer the location of the pack and unpack annotations (our □ and **let**□) in a program.[10] In Section 9.2 of that paper, the authors extend their system to include class constraints, much as we allow our existential packages to carry linear constraints.

Accordingly, EDWL would work well for us here and remove the need for these annotations. The EDWL algorithm is only a small change on the way some types are treated during bidirectional type-checking. Though the presentation of linear constraints is not written using a bidirectional algorithm, our implementation in GHC is indeed bidirectional (as GHC's existing type inference algorithm is bidirectional, as described by Peyton Jones et al. [2007] and Eisenberg et al. [2016]) and produces constraints much like we have presented here, formally. None of this would change in adapting EDWL. Indeed, it would seem that the two extensions are orthogonal in implementation, though avoiding the need for explicit packing and unpacking would make linear constraints easier to use.

9 RELATED WORK

OutsideIn. Our aim is to integrate the present work in GHC, and accordingly the qualified type system in Section 5 and the constraint inference algorithm in Section 6 follow a similar presentation to that of OutsideIn [Vytiniotis et al. 2011], GHC's constraint solver algorithm. Even though our presentation is self-contained, we outline some of the differences from that work.

The solver judgement in OutsideIn takes the following form:

$$Q\ ;\ Q_{given}\ ;\ \overline{\alpha}_{tch} \overset{solv}{\mapsto} C_{wanted} \rightsquigarrow Q_{residual}\ ;\ \theta$$

[10] Actually, Eisenberg et al. [2021] use an **open** construct instead of **let**□ to access the contents of an existential package, but that distinction does not affect our usage of existentials with linear constraints.

The main differences between OutsideIn's solver judgement and our solver judgements in Section 6.3 are:

- OutsideIn's judgement includes top-level axioms schemes separately (Q), which we have omitted for the sake of brevity and are instead included in Q_{given}.
- We present the *given* constraints (Q_{given} in OutsideIn) as two separate constraint sets U and L, standing for the unrestricted and linear parts respectively.
- In addition to constraint inference, OutsideIn performs type inference, requiring additional bookkeeping in the solver judgment. The solver takes as input a set of *touchable* variables $\overline{\alpha}_{tch}$ which record the type variables that can be unified at any given time, and produces a type substitution θ as an output. As discussed in Section 6, we do not perform type inference, only constraint inference. Therefore, our solver need not return a type assignment.
- Both OutsideIn and our solver output a set of constraints, $Q_{residual}$ and L_o respectively. However, the meaning of these contexts is different. OutsideIn's *residual* constraints $Q_{residual}$ correspond to the part of C_{wanted} that could not be solved from the assumptions. These residuals are then quantified over in the generalisation step of the inference algorithm. We omit these residuals, which means that our algorithm cannot infer qualified types. Our *output* constraints L_o instead correspond to the part of the *linear* givens L_i that were not used in the solution for C_w.
- Finally, while OutsideIn has a single kind of conjunction, our constraint language requires two: $Q_1 \otimes Q_2$ and $Q_1 \,\&\, Q_2$. This shows up when generating constraints for **case** expressions in the rule G-Case rule. OutsideIn accumulates constraints across branches (taking the union of each branch), whereas we need to make sure that each branch of a **case**-expression consumes the same constraints.

Ownership. Ownership and borrowing are the key features of Rust's safe memory management model. In Section 4 we show how linear constraints can be used to implement such an ownership model as a library. Although linear constraints do not have the convenience of Rust's syntax, we expect that they will support a greater variety of abstractions.

Clean is another language with built-in ownership typing. Like Haskell it is a lazy language. Mutation is performed by returning a new reference, like in Linear Haskell without linear constraints.

Languages with capabilities. The idea of using capabilities to enforce high-level resource usage protocols is not new [DeLine and Fähndrich 2001], and as such has been applied in practical programming languages before. Both Mezzo [Pottier and Protzenko 2013] and ATS [Zhu and Xi 2005] served as inspiration for the design of linear constraints. Of the two, Mezzo is more specialised, being entirely built around its system of capabilities. ATS is the closest to our system because it appeals explicitly to linear logic, and because the capabilities (known as *stateful views*) are not tied to a particular use case. However, ATS does not have full inference of capabilities.

Other than that, the two systems have a lot of similarities. They have a finer-grained capability system than is expressible in Rust (or our encoding of it in Section 4) which makes it possible to change the type of a reference cell upon write (though linear constraints could be used to implement such type-changing references too). They also eschew scoped borrowing in favour of more traditional read and write capabilities.

Linear constraints are more general than either Mezzo or ATS, while maintaining a considerably simpler inference algorithm, and at the same time supporting a richer set of constraints (such as GADTs). This simplicity is a benefit of abstracting over the simple-constraint domain. In fact, it should be possible to see Mezzo or ATS as particular instantiations of the simple-constraint domain, with linear constraints providing the general inference mechanism.

Linearly typed languages. Affe [Radanne et al. 2020] is a linearly typed ML-style core language with mutable references and arrays, augmented with a notion of borrowing. It has dedicated syntax for the scope of borrows. In contrast, we represent scopes as functions. Affe is presented as a fully integrated solution, while linear constraints is a small layer on top of Linear Haskell.

Logic programming. There are a lot of commonalities between GHC's constraint and logic programs. Traditional type classes can be seen as Horn clause programs, much like Prolog programs. GHC puts further restrictions in order to avoid backtracking for speed and predictability.

The recent addition of quantified constraints [Bottu et al. 2017] extends type class resolution to Hereditary Harrop [Miller et al. 1987] programs. A generalisation of the Hereditary Harrop fragment to linear logic, described by Hodas and Miller [1994], is the foundation of the Lolli language [Hodas 1994]. The authors also coin the notion of *uniform* proof. A fragment where uniform proofs are complete supports goal-oriented proof search, like Prolog does.

Completeness of uniform proofs is equivalent to Lemma 6.1, which, in turn, is used in the proof of the soundness Lemma 6.4. Therefore our linear constraints are compatible with quantified constraints: we simply need to adapt Lemma 6.1.

It is interesting that goal-oriented search is baked into the definition of OutsideIn. It's not only used as the constraint solving strategy, but it seems to required for the soundness of the constraint generation algorithm. Or, if they are not required, uniform proofs are at least an effective strategy to prove soundness.

10 CONCLUSION

We showed how a simple linear type system like that of Linear Haskell can be extended with an inference mechanism which lets the compiler manage some of the additional complexity of linear types instead of the programmer. Linear constraints narrow the gap between linearly typed languages and dedicated linear-like typing disciplines such as Rust's, Mezzo's, or ATS's.

ACKNOWLEDGMENTS

Jean-Philippe Bernardy is supported by grant 2014-39 from the Swedish Research Council, which funds the Centre for Linguistic Theory and Studies in Probability (CLASP) in the Department of Philosophy, Linguistics, and Theory of Science at the University of Gothenburg. Nicolas Wu is supported by EPSRC Grant EP/S028129/1.

REFERENCES

Andreas Abel and Jean-Philippe Bernardy. 2020. A unified view of modalities in type systems. *Proceedings of the ACM on Programming Languages* 4, ICFP (2020).

Jean-Philippe Bernardy, Mathieu Boespflug, Ryan R. Newton, Simon Peyton Jones, and Arnaud Spiwack. 2017. Linear Haskell: Practical Linearity in a Higher-order Polymorphic Language. *Proc. ACM Program. Lang.* 2, POPL, Article 5 (Dec. 2017), 29 pages. https://doi.org/10.1145/3158093

Gert-Jan Bottu, Georgios Karachalias, Tom Schrijvers, Bruno C. d. S. Oliveira, and Philip Wadler. 2017. Quantified Class Constraints. In *Proceedings of the 10th ACM SIGPLAN International Symposium on Haskell* (Oxford, UK) *(Haskell 2017)*. Association for Computing Machinery, New York, NY, USA, 148–161. https://doi.org/10.1145/3122955.3122967

Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. 2000. Efficient resource management for linear logic proof search. *Theoretical Computer Science* 232, 1 (2000), 133 – 163. https://doi.org/10.1016/S0304-3975(99)00173-5

Karl Crary, David Walker, and J. Gregory Morrisett. 1999. Typed Memory Management in a Calculus of Capabilities. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 262–275. https://doi.org/10.1145/292540.292564

Vincent Danos, Jean Baptiste Joinet, and Harold Schellinx. 1993. The structure of exponentials: Uncovering the dynamics of linear logic proofs. In *Computational Logic and Proof Theory*, Georg Gottlob, Alexander Leitsch, and Daniele Mundici (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 159–171.

Robert DeLine and Manuel Fähndrich. 2001. Enforcing High-Level Protocols in Low-Level Software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, Michael Burke and Mary Lou Soffa (Eds.). ACM, 59–69. https://doi.org/10.1145/378795.378811

Facundo Domínguez. 2020. Safe memory management in inline-java using linear types. (2020). https://web.archive.org/web/20200926082552/https://www.tweag.io/blog/2020-02-06-safe-inline-java/ Blog post.

Richard A. Eisenberg, Guillaume Duboc, Stephanie Weirich, and Daniel Lee. 2021. An Existential Crisis Resolved: Type inference for first-class existential types. *Proc. ACM Program. Lang.* 5, ICFP (2021).

Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan G. Ahmed. 2016. Visible Type Application. In *Programming Languages and Systems*, Peter Thiemann (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 229–254.

Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. 2006. Linear Regions Are All You Need. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27-28, 2006, Proceedings (Lecture Notes in Computer Science, Vol. 3924)*, Peter Sestoft (Ed.). Springer, 7–21. https://doi.org/10.1007/11693024_2

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science* 50, 1 (1987), 1–101. https://doi.org/10.1016/0304-3975(87)90045-4

Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. 1996. Type Classes in Haskell. *ACM Trans. Program. Lang. Syst.* 18, 2 (March 1996).

J.S. Hodas and D. Miller. 1994. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation* 110, 2 (1994), 327 – 365. https://doi.org/inco.1994.1036

Joshua Seth Hodas. 1994. Logic programming in intuitionistic linear logic: Theory, design, and implementation. https://repository.upenn.edu/dissertations/AAI9427546

Mark P. Jones. 1994. A theory of qualified types. *Science of Computer Programming* 22, 3 (1994), 231 – 256. https://doi.org/10.1016/0167-6423(94)00005-0

Csongor Kiss, Arnaud Spiwack, Jean-Philippe Bernardy, Nicolas Wu, and Richard A. Eisenberg. 2022. Prototype implementation of linear constraints in GHC. http://archive.softwareheritage.org/swh:1:rev:f6fc5ba23770b42d1d6020e177787757b16a9ea0;origin=https://github.com/kcsongor/ghc;visit=swh:1:snp:aa61d803eaec9eb4425e3eb8ed2b0fbbd60633cc. Code fragment.

Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the Pi-Calculus. *ACM Trans. Program. Lang. Syst.* 21, 5 (Sept. 1999), 914–947. https://doi.org/10.1145/330249.330251

Kazutaka Matsuda. 2020. Modular Inference of Linear Types for Multiplicity-Annotated Arrows. In *Programming Languages and Systems*, Peter Müller (Ed.). Springer International Publishing, Cham, 456–483.

Dale A. Miller, Gopalan Nadathur, and Andre Scedrov. 1987. Hereditary Harrop Formulas and Uniform Proof Systems. In *Proceedings of the Second Annual IEEE Symposium on Logic in Computer Science (LICS 1987)* (Ithaca, NY, USA). IEEE Computer Society Press, 98–105.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. 2007. Practical type inference for arbitrary-rank types. *Journal of Functional Programming* 17, 1 (2007), 1–82. https://doi.org/10.1017/S0956796806006034

Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press, Cambridge, MA.

François Pottier and Jonathan Protzenko. 2013. Programming with Permissions in Mezzo. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 173–184. https://doi.org/10.1145/2500365.2500598

François Pottier and Didier Rémy. 2005. The essence of ML type inference. (2005).

Gabriel Radanne, Hannes Saffrich, and Peter Thiemann. 2020. Kindly Bent to Free Us. *Proc. ACM Program. Lang.* 4, ICFP, Article 103 (Aug. 2020), 29 pages. https://doi.org/10.1145/3408985

Michael Shulman. 2018. Linear logic for constructive mathematics. arXiv:1805.07518 [math.LO]

Frederick Smith, David Walker, and J. Gregory Morrisett. 2000. Alias Types. In *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1782)*, Gert Smolka (Ed.). Springer, 366–381. https://doi.org/10.1007/3-540-46425-5_24

Arnaud Spiwack, Csongor Kiss, Jean-Philippe Bernardy, Nicolas Wu, and Richard A. Eisenberg. 2021. Linear Constraints. *CoRR* abs/2103.06127 (2021). arXiv:2103.06127 https://arxiv.org/abs/2103.06127

Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2007. System F with Type Equality Coercions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation* (Nice, Nice, France) *(TLDI '07)*. Association for Computing Machinery, New York, NY, USA, 53–66. https://doi.org/10.1145/1190315.1190324

Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. 2010. Let Should Not Be Generalized. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Madrid, Spain) *(TLDI '10)*. Association for Computing Machinery, New York, NY, USA, 39–50. https://doi.org/10.1145/1708016.1708023

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(X) Modular type inference with local assumptions. *Journal of Functional Programming* 21, 4-5 (2011), 333–412. https://doi.org/10.1017/S0956796811000098

David Walker and J. Gregory Morrisett. 2000. Alias Types for Recursive Data Structures. In *Types in Compilation, Third International Workshop, TIC 2000, Montreal, Canada, September 21, 2000, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 2071)*, Robert Harper (Ed.). Springer, 177–206. https://doi.org/10.1007/3-540-45332-6_7

Stephanie Weirich, Antoine Voizard, Pedro Henrique Azevedo de Amorim, and Richard A. Eisenberg. 2017. A Specification for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 1, ICFP, Article 31 (Aug. 2017), 29 pages. https://doi.org/10.1145/3110275

Dengping Zhu and Hongwei Xi. 2005. Safe Programming with Pointers Through Stateful Views. In *Practical Aspects of Declarative Languages*, Manuel V. Hermenegildo and Daniel Cabeza (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 83–97.