Linear Constraints

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

A linear argument must be consumed exactly once in the body of its function. A linear type system can verify the correct usage of resources such as file handles and manually managed memory. But this verification 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 to be filled in automatically by the compiler. Linear constraints are presented as a qualified type system, together with an inference algorithm which extends OutsideIn, 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, laziness, linear logic, linear types, constraints, inference

ACM Reference Format:
Jean-Philippe Bernardy, Richard A. Eisenberg, Csongor Kiss, Arnaud Spiwack, and Nicolas Wu. 2021. Linear Constraints. In *Proceedings of ICFP (ICFP'21).* ACM, New York, NY, USA, 32 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Linear type systems have seen a renaissance in recent years in various mainstream 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. In fact, there seems to be a tradeoff between these two aspects. Rust has a domain-specific *borrow-checker* that aids the programmer in reasoning about memory ownership, but it doesn't naturally scale beyond this use case. Linear Haskell, on the other hand, supports general purpose linear types, but using them to emulate Rust's ownership model is a painful exercise, because the compiler doesn't know how to help, requiring the programmer to carefully thread resource tokens.

To get a sense of the power and the pain of using linear types, consider the following example from Bernardy et al. [2017][1]:

```
firstLine :: FilePath → IO_L String
firstLine fp = do { h ← openFile fp
                  ; (h, Ur xs) ← readLine h
                  ; closeFile h
                  ; return xs }
```

This simple function opens a file, reads its first line, then closes it. Linearity ensures that the file handle h is consumed at the end.

[1] IO_L is the linear IO monad

ICFP'21, 2021,
2021. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Forgetting to call *closeFile h* would result in a type error since *h* would remain unused at the end of the function. Notice that *readLine* consumes the file handle, and returns a fresh *h* that shadows the previous version, to be used in further interactions with the file. The line's content is a string *xs* that is returned in an *Ur* wrapper (pronounced "unrestricted") to signify that it can be used arbitrary many times. Compare this function with what one would write in a non-linear language:

```
firstLine :: FilePath → IO String
firstLine fp = do { h ← openFile fp
                  ; xs ← readLine h
                  ; closeFile h
                  ; return xs }
```

This style is less safe, because the type system does not keep track of the file handle. But it is also simpler. It would seem that using a linear language means trading clarity for safety. When modelling a handle as a linear resource, the type system must know at all times where it is being consumed, so the file handle is passed around manually, resulting in extra noise. But when reading the non-linear version, it is clear where the handle is used, and ultimately, consumed. Couldn't the compiler figure this out without extra help?

Rust introduces the borrow checker for this very purpose. But it turns out that all you need is to generalise Haskell's type class constraints to *linear constraints*. Like class constraints, linear constraints are passed around implicity by the compiler, and like linear arguments, they can be used to track resources such as file handles. The beauty of linear constraints is their simplicity: all we need to do, really, is to pull together the threads from four papers ([Bernardy et al. 2017; Cervesato et al. 2000; Hodas and Miller 1994; Vytiniotis et al. 2011]).

With this extension, the final version of *firstLine* is almost the same as the traditional version above, with a few minor modifications:

```
firstLine :: FilePath → IO_L String
firstLine fp = do {pack! h ← openFile fp
                  ; pack! xs ← readLine h
                  ; closeFile h
                  ; return xs }
```

The only changes from the unsafe version are that this version runs in the linear IO monad, and explicit **pack!** annotations are used to indicate the variables that require special treatment. Crucially, the resource representing the open file no longer needs to be passed around manually.

Our contributions are as follows:

- A system of qualified types that allows a constraint assumption to be given a multiplicity. Linear assumptions are used precisely once in the body of a definition (Section 4).
- This system supports examples that have motivated the design of several resource-aware systems, such as ownership à la Rust (Section 3), or capabilities in the style of Mezzo [Pottier and Protzenko 2013] or ATS [Zhu and Xi 2005]; accordingly, our system may serve as a way to unify these lines of research.
- An inference algorithm, that respects the multiplicity of assumptions. We prove that this algorithm is sound with respect to our type system (Section 5).
- Expressions in our qualified type system desugar into a core language (directly adapted from Linear Haskell [Bernardy et al. 2017]) that supports linear functions. We prove that the output of desugaring is well-typed in the core language (Section 6).

$openFile :: FilePath \to IO_L\ Handle$
$readLine :: Handle \multimap IO_L\ (Handle, Ur\ String)$
$closeFile :: Handle \multimap IO_L\ ()$

(a) Linear Types

$openFile :: FilePath \to IO_L\ (\exists\ h.Ur\ (Handle\ h) \circ\!\!= Open\ h)$
$readLine :: Open\ h \Rightarrow\!\circ Handle\ h \to IO_L\ (Ur\ String \circ\!\!= Open\ h)$
$closeFile :: Open\ h \Rightarrow\!\circ Handle\ h \to IO_L\ ()$

(b) Linear Constraints

Fig. 1. Interfaces for file manipulation

2 MOTIVATION

The goal of this paper is to introduce linear constraints as a mechanism for ergonomically using linear types in Haskell. In this section, we sketch out some of the basics of how linear types work in Haskell, and then give a number of examples that motivate our extension into linear constraints.

2.1 Linear Types

Linear Haskell [Bernardy et al. 2017] adds a new type of functions, dubbed *linear functions*, and written $a \multimap b$[2]. A linear function consumes its argument exactly once. More precisely, Linear Haskell [Bernardy et al. 2017, Section 2.1] defines linear functions as:

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

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

Note that a linear arrow specifies *how the function uses its argument*. It does not restrict *the arguments to which the function can be applied*. 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:

$g :: a \to b$
$g\ x = f\ x$

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

The example in the previous section, when calling *readLine*, consumes the linear handle h created by *openFile*. Therefore h can no longer be used to close the file: doing so would result in a type error. To resolve this, a new handle for the same file is produced that can be used with *closeFile*. In the code, the same name h is given to the new handle, thus shadowing the old h that can no longer be used anyway, and to give the illusion that it is being passed around.

From the perspective of the programmer, this is unwanted boilerplate. Using shadowing is a trick that hides an underlying problem that should not exist: it is not the handle that should be

[2]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.

consumed by *readLine* at all. Rather, it is its implicit state as a handle for an open file that should be consumed exactly once. Once this open state is consumed, the handle can no longer be read from or closed without triggering a compile time error.

2.2 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 fat arrow $\Rightarrow$. Everything to the left of a fat arrow is called a *constraint*. Here $Show\ a$ is a type-class constraint, but there are other kinds of constraints such as equality constraints or implicit parameter constraints.

What is crucial, for our purpose, is that constraints are handled implicitly by the typechecker. That is, if we want to *show* an integer we would write *show* 42, and the typechecker would handle proving that *Show Int* holds without intervention from the programmer. Thus, constraints are a convenient mechanism that allow the compiler to automatically fill in implicit arguments.

For our *readLine* example, the implicit argument is the state of a handle h. This can be managed as a constraint called $Open\ h$, which indicates that the file associated with the handle is open. The idea is that this constraint should be a linear value, so that it can only be consumed exactly once. In order to manage linearity implicitly, this article introduces a linear fat arrow $\multimap\!\!\Rightarrow$, much like Linear Haskell introduced a linear function arrow $\multimap$. We dub constraints to the left of a linear fat arrow *linear constraints*. Using the linear constraint $Open\ h$ to represent that a file is open, we can give the following type to *closeFile*:

$closeFile :: Open\ h \multimap\!\!\Rightarrow Handle\ h \to IO_L\ ()$

There are a few things to notice:

- First, there is now a type variable h. In contrast, the version in Figure 1a without linear constraints has type $closeFile :: Handle \multimap IO_L\ ()$. The type variable h is a type-level name for the file handle which we are closing. Ideally, we'd like the linear constraint to refer directly to the handle value, and have the type $closeFile :: (h :: Handle) \to Open\ h \multimap\!\!\Rightarrow IO_L\ ()$. While giving a name to a function argument is the bread and butter of more dependently typed languages such as ATS [Xi 2017] or Liquid Haskell [Vazou et al. 2013], Haskell doesn't have such a naming mechanism built in yet, so we have to resort to a type variable to mediate the link between handles and linear constraints.
- Second, if we have a single, linear, $Open\ h$ available, then after *closeFile* there will not be any $Open\ h$ left to use, thus preventing the file from being closed twice. This is precisely what we were trying to achieve.

This deals with closing files and ensuring that a handle cannot be closed more than once. However, we still need to explain how a constraint $Open\ h$ can come into existence. For this, we introduce a type construction $\exists\ a_1 \ldots a_n.t \circ\!\!= Q$, where Q is some (linear) constraint that is created. The type of *openFile* is thus:

$openFile :: FilePath \to IO_L\ (\exists\ h.Ur\ (Handle\ h) \circ\!\!= Open\ h)$

The output of this function is a new handle that is unrestricted, along with a new constraint that indicates that the handle is open. Since the handle is unrestricted it can be freely passed around. Then, the handle's state is implicitly tracked by the $Open\ h$ constraint.

We must also ensure that *readLine* can both promise to only operate on an open file, and to keep that file open after reading. To do so, its signature indicates that given a handle h, it consumes and produces the implicit *Open h* constraint, while also producing an unrestricted *String*.

$$readLine :: Open\ h \multimap Handle\ h \to IO_L\ (Ur\ String \circ\!\!= Open\ h)$$

The fact that existential quantification generates new type-level names is folklore. It's used crucially in the interface of the *ST* monad [Launchbury and Peyton Jones 1994] and in type-class reflection [Kiselyov and Shan 2004] (in both of these cases, existential quantification is encoded as rank-2 universal quantification). We shall use it in exactly this way: *openFile* uses an existential quantifier to generate the type-level name of the file handle. Existentially quantified types are paired with a constraint Q which we understand as being returned by functions. We will freely omit the $\exists\ a_1 \ldots a_n.$ or $\circ\!\!= Q$ parts when they are empty.

Haskell doesn't have an existential quantifier, however it can be encoded as a GADT. For instance $\exists\ h.Ur\ (Handle\ h) \circ\!\!= Open\ h$ can be implemented as

data *PackHandle* **where**
  $Pack :: Open\ h \multimap Handle\ h \to PackHandle$

That is, while this paper introduces special-purpose existential types, in the implementation as a GHC extension, packed linear constraints will piggy-back on the standard GADT syntax. Correspondingly, existential types are introduced by a data constructor, which we write as **pack**.

When pattern-matching on a **pack** all the existentially quantified names are introduced in scope and all the returned constraints are made available. We also use **pack**! x as a shorthand for **pack** $(Ur\ x)$. With all these ingredients we can indeed write the example given in the introduction.

2.3 Motivating Examples

To get a sense of how the features we introduce should behave, let's look at some simple examples. Using constraints to represent expendable 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 class C that is consumed by the *useC* function.

class C

$useC :: C \multimap Int$

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

2.3.1 Dithering. Now consider the following program, which we reject:

$dithering :: C \multimap Bool \to Int$
$dithering\ x =$ **if** x **then** $useC$ **else** 10

The problem with *dithering* is that it does not commit properly to how much resource it requires: the branch where x holds will use the resource C whereas the other does not.

2.3.2 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 multiplicity of the second arrow is ω.

One way to improperly use the linear *const* is by neglecting a linear variable:

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

The problem with *neglecting* is that although *useC* is mentioned in this program, it is never evaluated because *const* does not use its second argument. The constraint C is thus consumed ω times so this program ought to be rejected.

The rule is that a linear constraint is only consumed in a linear context. For example,

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

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

2.3.3 Overusing. Finally, consider the following program, which should be rejected:

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

This is program is rejected because it uses C twice. However, the following version is accepted:

$$notOverusing :: (C, C) \multimap (Int, Int)$$
$$notOverusing = (useC, useC)$$

That is, it is possible to have multiple copies of a given constraint. Our system is designed so that the order of resolution is non-deterministic. This corresponds to the assumption that all instances of a linear constraint are equivalent. Consequently, if an API uses linear constraints, it must be designed to ensure that the runtime behaviour is not dependent on the order of constraint resolution.

2.4 Linear IO

The file handling example discussed in sections 2.1 and 2.2 uses a linear version of the *IO* monad, IO_L. There are two main modifications compared to the traditional *IO*. Firstly, the type of the monadic operations $\gg=$ and *return* are changed to use linear arrows.

$$(\gg=) :: IO_L\ a \multimap (a \multimap IO_L\ b) \multimap IO_L\ b$$
$$return :: a \multimap IO_L\ a$$

Bind must be linear because, as explained in the previous section, a linear constraint can only be consumed in a linear context. For the following program to typecheck, bind must be linear:

$$\begin{aligned} &readTwo :: Open\ h \multimap Handle\ h \to IO_L\ (Ur\ (String, String) \circ\!\!- Open\ h) \\ &readTwo\ h = readLine\ h \gg= \lambda \mathbf{case}\ \mathbf{pack}!\ xs \to \\ &\qquad\qquad readLine\ h \gg= \lambda \mathbf{case}\ \mathbf{pack}!\ ys \to \\ &\qquad\qquad return\ (\mathbf{pack}!\ (xs, ys)) \end{aligned}$$

If it were not linear, the first argument *readLine h* of the first bind would not be able to consume the *Open h* constraint.

3 APPLICATION: MEMORY OWNERSHIP

Let us now turn to a more substantial example: manual memory management. In Haskell, memory deallocation is normally the responsibility of a garbage collector. However, garbage collection comes with runtime costs that not all applications can afford. Another realistic scenario is to imagine a Haskell program calling foreign functions that reside in a different runtime system [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).

3.1 Ownership constraints

With linear constraints, it is possible to give a *pure* interface for manual memory management, which enforces correct deallocation of memory. The approach is to use a linear constraint to represent *ownership* of a memory location in the style of Rust, *i.e.* the responsibility of deallocating it. We use a linear constraint *Own n*, such that a program which has an *Own n* constraint in context is responsible of deallocating the memory associated with a memory location *n*. Because of linearity, this constraint must be consumed exactly once, so it is guaranteed that the memory is deallocated correctly. In the above, *n* is a type variable (of a special kind *Location*) which represents a memory location. Locations mediate the relationship between references and ownership constraints.

For more granular control, we will be using three linear constraints: one for reading, one for writing, and one for ownership.

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

Thanks to this extended set of constraints, programs are able to read from or write to a memory reference without owning it. 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. Rust disallows mutable aliasing for the same reason: ensuring that writes cannot be observed through other references is what allows treating mutable structures as "pure". Therefore, writing also requires owning the read capability. We systematically use the *RW* set of constraints, defined below, instead of *Write*.

type *RW n* = (*Read n*, *Write n*)

Likewise, a location cannot be deallocated if any part of the program has a read or write reference to it, so all 3 capabilities are needed for ownership. So we use *O*, instead of *Own*, defined thus:

type *O n* = (*Read n*, *Write n*, *Own n*)

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

$$newRef :: (\forall\ n.\ O\ n \multimap AtomRef\ a\ n \to Ur\ b) \multimap Ur\ b$$

The reference is made available in a function which we call the *scope* of the reference. The return type of the scope is, crucially, *Ur b*. Indeed, if we would allow returning any type *b* then the *O n* constraint could be embedded in *b*, and therefore escape from the scope. This would effectively let *O n* be an unrestricted constraint, and no longer guarantee that the reference has a unique owner.

To read a reference, a simple *Read* constraint is demanded, and immediately returned. Writing is handled similarly.

$$readRef :: (Read\ n) \multimap AtomRef\ a\ n \to Ur\ a \circ\!\!- Read\ n$$
$$writeRef :: (RW\ n) \multimap AtomRef\ a\ n \to a \to () \circ\!\!- RW\ n$$

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*. This is ensured by the combination of the language and library behaviour. For example, here is how to write two values (*a* and *b*) to the same reference *x*:

```
case writeRef x a of
  pack _ → case writeRef x b of
    pack _ → ...
```

The language semantics forces the programmer to do case analysis to access the returned *Write* constraints, and *writeRef* must be strict in the *Write* constraint that it consumes.

Deallocation consumes all linear constraints associated with $O\ n$.

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

As an alternative to freeing the reference, one could transfer control of the memory location to the garbage collector. This operation is sometimes called "freezing":

$$freezeRef :: O\ n \multimap AtomRef\ a\ n \to Ur\ a$$

3.2 Arrays

The toolkit we set up handles references to base types just fine. But what about references to references; or, indeed, arrays of arrays? A common example of plain Linear Haskell is a pure interface to mutable arrays [Bernardy et al. 2017, Section 2.2]. There we have two types, $MArray\ a$, used linearly, for mutable arrays, and $Array\ a$, used unrestrictedly, for immutable arrays. Mutable arrays can be frozen using $freeze :: MArray\ a \multimap Ur\ (Array\ a)$. Note that this version of *freeze* can not support mutable arrays of mutable arrays. The core of the issue is that mutable arrays and immutable arrays have different types.

The ownership API, on the other hand, is readily extended to nested mutable arrays. The key ingredient is for inner arrays to relinquish their ownership to the outer array.

$$\textbf{data}\ PArray\ (a :: Location \to Type)\ (n :: Location)$$
$$newPArray :: (\forall\ n.\ O\ n \multimap PArray\ a\ n \to Ur\ b) \multimap Ur\ b$$

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.

When writing a reference (be it an array or an *AtomRef*) in an array, ownership of the reference is relinquished to the array.

$$writePArray :: (RW\ n, O\ p) \multimap PArray\ a\ n \to Int \to a\ p \to () \circ\!\!= RW\ n$$

More precisely, the ownership of the location p is absorbed into that of n. Therefore, the associated operational semantics is to move the reference inside the array (and deallocate any previous reference at that index).

We still want to have read and write access to inner references, of course, so we use *lendPArrayElt*:

$$lendPArrayElt :: RW\ n \multimap PArray\ a\ n \to Int \to (\forall\ p.\ RW\ p \multimap a\ p \to r \circ\!\!= RW\ p) \multimap r \circ\!\!= RW\ n$$

The $lendPArrayElt\ a\ i\ k$ primitive lends access to the reference at index i in a, to a scope function k (in Rust terminology, the scope "borrows" an element of the array). Here, the return type of the scope, r, is not in Ur: since the scope must return the $RW\ p$ constraint, it is not possible to leak it out by packing it into r, so it's not necessary to wrap the result in Ur. Crucially, with this API, $RW\ n$ and $RW\ p$ are never simultaneously available. The following function would not be sound:

$$extractEltWrong :: RW\ n \multimap PArray\ a\ n \to Int \to \exists\ p.Ur\ (a\ p) \circ\!\!=\!(RW\ n, RW\ p)$$

Indeed, when called from a context where the program owns n, n could immediately be deallocated, even though $RW\ p$ would remain available, letting us write to free memory. For the same reason, gaining read access to an element needs to be done using a scoped API as well:

$$\begin{aligned} \mathit{lendPArrayEltRead} \;::\; & \mathit{Read}\ n \multimap \mathit{PArray}\ a\ n \to \mathit{Int} \\ & \to (\forall\ p.\, \mathit{Read}\ p \multimap a\ p \to r \circ\!\!- \mathit{Read}\ p) \\ & \multimap r \circ\!\!- \mathit{Read}\ n \end{aligned}$$

Finally, we can freeze arrays, and arrays of arrays or more, using the following primitive:

$\mathit{freezePArray} :: O\ n \multimap \mathit{PArray}\ a\ n \to () \Leftarrow \mathit{Read}\ n$

After $\mathit{freezePArray}\ n$, we have unrestricted read access to n (and any element of n), as expected.

$\mathit{readArray} :: \mathit{Read}\ n \Rightarrow \mathit{PArray}\ a\ n \to \mathit{Int} \to a\ n$

Crucially, and in contrast to the plain linear types API the type *PArray* is both the type of mutable arrays and immutable arrays, *freezePArray* only changes the permissions. And since permissions are linear constraints, this is all managed automatically by the compiler.

4 A QUALIFIED TYPE SYSTEM

Let us now describe the technical material which supports the examples we explored above. The presentation in this section, as well as Section 5, strongly mirrors the presentation of OutsideIn [Vytiniotis et al. 2011]. OutsideIn is the foundation of the type inference algorithm of GHC. By building from this base, we can highlight the concrete changes required in GHC to support linear constraints. However, we have chosen, for the sake of clarity, to omit details of OutsideIn which do not interact meaningfully with linear constraints. We shall point out such simplifications as they arise.

4.1 Multiplicities

Like in Linear Haskell [Bernardy et al. 2017] we make use of a system of *multiplicities*, which describe how many times a function consumes its input. Linear Haskell additionally supports multiplicity polymorphism. For our purposes, we need only the simplest system of multiplicity: that composed of only 1 (representing linear functions) and ω (representing regular Haskell functions).

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

The idea of multiplicity goes back at least to Ghica and Smith [2014], where it is dubbed a *resource semiring*. The power of multiplicities is that they can encode the structural rules of linear logic with only semiring operation—addition and multiplication[3].

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

We do not support multiplicity polymorphism on constraint arguments, and we believe that such a feature would not be very useful. Multiplicity polymorphism of regular function arguments is used to avoid duplicating the definition of higher-order functions. The poster child is $\mathit{map} :: (a \to_m b) \to [a] \to_m [b]$, 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 function as explained in Section 2.1. 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 5.3 to support multiplicity-polymorphic constraints.

[3]We adopt the convention that equations defining a function by pattern matching are marked with a { to their left

4.2 Simple constraints

Following OutsideIn [Vytiniotis et al. 2011, Section 3.2, in particular Figure 3], we parameterise our entire type system by a constraint domain: the X in OutsideIn(X). Such a domain is characterised by a set of *atomic constraints*, written $\mathbf{q}$, and an entailment relation $Q_1 \Vdash Q_2$, with properties specified in Figure 2. We adopt GHC's terminology and call these *simple constraints* to distinguish them from the richer constraints of Section 5.1.

For instance, in GHC, the domain includes type classes, and the entailment relation describes instance resolution. However, our needs here are more modest: atomic constraints uninterpreted by the entailment relation are sufficient. Parameterising over the domain therefore only serves to support the rest of Haskell, or future extensions.

$$Q \quad ::= \quad \pi\cdot\mathbf{q} \mid Q_1 \otimes Q_2 \mid \varepsilon \quad \text{Simple constraints}$$

The multiplicity π in front of atomic constraints is the *scaling factor*. It indicates whether the constraint is to be used linearly (1) or without restriction (ω).

Atomic constraints are treated abstractly by our system (just like they are in OutsideIn). For inference, in Section 5, we will need a domain-specific solver, of which we only require that it adheres to the interface given in Section 5.3. But for the sake of this section, we only need that the domain be equipped with the entailment relation.

The empty constraint is written as ε. This corresponds to the multiplicative truth $\mathbf{1}$ of linear logic, but we chose this notation because it is more distinct from the 1 multiplicity.

OutsideIn introduces, as part of the constraint domain, a generalised kind of constraint $\boldsymbol{Q}$, which include top-level axioms, such as type-class instance declarations. Such top-level axioms are never linear – just like how top-level definitions are never linear in Linear Haskell [Bernardy et al. 2017] – and as such they don't have interesting interactions with the rest of the system, and we choose to omit them for simplicity.

We consider simple constraints to be equal up to associativity and commutativity of tensor products, as well as idempotence of the unrestricted constraints. We require the entailment relation to respect these properties. That is:

$$\begin{aligned} Q_1 \otimes Q_2 &= Q_2 \otimes Q_1 \\ (Q_1 \otimes Q_2) \otimes Q_3 &= Q_1 \otimes (Q_2 \otimes Q_3) \\ \omega\cdot\mathbf{q} \otimes \omega\cdot\mathbf{q} &= \omega\cdot\mathbf{q} \\ Q \otimes \varepsilon &= Q \end{aligned}$$

Scaling is extended to all constraints as follows:

$$\begin{cases} \pi\cdot\varepsilon &= \varepsilon \\ \pi\cdot(Q_1 \otimes Q_2) &= \pi\cdot Q_1 \otimes \pi\cdot Q_2 \\ \pi\cdot(\rho\cdot Q) &= (\pi\cdot\rho)\cdot Q \end{cases}$$

The rule that $\omega\cdot(Q_1 \otimes Q_2) = \omega\cdot Q_1 \otimes \omega\cdot Q_2$ is not a typical feature of linear logic. Linear Haskell, however, introduces the corresponding type isomorphism for the sake of polymorphism. While this article isn't concerned with polymorphism, this equality does make the overall presentation a bit simpler. Note, in particular, that $1\cdot Q = Q$ and $\omega\cdot Q \otimes \omega\cdot Q = \omega\cdot Q$.

We will often omit the scaling factor for linear atomic constraints and write $\mathbf{q}$ for $1\cdot\mathbf{q}$.

Definition 4.1 (Requirements for the entailment relation). The constraint entailment relation must satisfy the properties in Figure 2.

Apart from linearity, the main difference with OutsideIn is that we don't require the presence of equality constraints. We come back to the motivation for this simplification in Section 5.

$Q \Vdash Q$
if $Q_1 \Vdash Q_2$ and $Q \otimes Q_2 \Vdash Q_3$ then $Q \otimes Q_1 \Vdash Q_3$
if $Q \Vdash Q_1 \otimes Q_2$ then there exists Q' and Q'' such that $Q = Q' \otimes Q''$, $Q' \Vdash Q_1$ and $Q'' \Vdash Q_2$
if $Q \Vdash \varepsilon$ then there exists Q' such that $Q = \omega \cdot Q'$
if $Q_1 \Vdash Q_1'$ and $Q_2 \Vdash Q_2'$ then $Q_1 \otimes Q_2 \Vdash Q_1' \otimes Q_2'$
if $Q \Vdash \rho \cdot \mathbf{q}$ then $\pi \cdot Q \Vdash (\pi \cdot \rho) \cdot \mathbf{q}$
if $Q \Vdash (\pi \cdot \rho) \cdot \mathbf{q}$ then there exists Q' such that $Q = \pi \cdot Q'$ and $Q' \Vdash \rho \cdot \mathbf{q}$
if $Q_1 \Vdash Q_2$ then $\omega \cdot Q_1 \Vdash Q_2$
if $Q_1 \Vdash Q_2$ then for all Q', $\omega \cdot Q' \otimes Q_1 \Vdash Q_2$

Fig. 2. Requirements for the entailent relation $Q_1 \Vdash Q_2$

a, b	::=	$\ldots$	Type variables
x, y	::=	$\ldots$	Expression variables
K	::=	$\ldots$	Data constructors
σ	::=	$\forall \overline{a}.Q \Rightarrow \tau$	Type schemes
τ, υ	::=	$a \mid \exists \overline{a}.\tau \multimap Q \mid \tau_1 \to_\pi \tau_2 \mid T\,\overline{\tau}$	Types
Γ, Δ	::=	$\bullet \mid \Gamma, x{:}_\pi \sigma$	Contexts
e	::=	$x \mid K \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{pack}\ e$	Expressions
	$\mid$	$\mathbf{unpack}\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{case}_\pi\ e\ \mathbf{of}\ \{\overline{K_i\ \overline{x_i} \to e_i}\}$	
	$\mid$	$\mathbf{let}_\pi\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{let}_\pi\ x : \sigma = e_1\ \mathbf{in}\ e_2$	

Fig. 3. Grammar of the qualified type system

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.

Lemma 4.2 (Promotion). *If $Q_1 \Vdash Q_2$, then $\pi \cdot Q_1 \Vdash \pi \cdot Q_2$.*

Lemma 4.3 (Inversion of promotion). *If $Q_1 \Vdash \pi \cdot Q_2$, then $Q_1 = \pi \cdot Q_1'$ and $Q_1' \Vdash Q_2$ for some Q'.*

Proofs of these lemmas (and others) appear in our anonymized supplementary material.

4.3 Typing rules

With this material in place, we can present our type system. The grammar is given in Figure 3, and the typing rules in Figure 4.

Like in OutsideIn [Vytiniotis et al. 2011, Section 4], the type system of Figure 4 is a *qualified type system* in the style first introduced by Jones [1994]. Such a qualified type system introduces a judgement of the form $Q; \Gamma \vdash e : \tau$, where Γ is a standard type context, and Q is a simple constraint as in Section 4.2. Q behaves much like Γ, which will be instrumental for desugaring in Section 6; the main difference is that Γ is addressed explicitly, whereas Q is used implicitly in rule E-Var.

The type system of Figure 4 is purely declarative: it doesn't have any algorithmic properties. We will see how to infer constraints in Section 5. Yet, this system is our ground truth: a system with a simple enough definition that programmers can reason about typing. 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 6.

The main differences with OutsideIn, either for simplification, or for linear constraints, are:

$\boxed{x{:}_1\sigma \in \Gamma}$ *(Context membership)*

$$\textsc{In-Var}\ \frac{}{x{:}_1\sigma \in x{:}_1\sigma} \qquad \textsc{In-Weaken}\ \frac{x{:}_1\sigma \in \Gamma}{x{:}_1\sigma \in \omega\cdot\Delta + \Gamma}$$

$\boxed{\pi\cdot\Gamma}$ *(Context scaling)*

$$\textsc{S-Empty}\ \frac{}{\pi\cdot\bullet = \bullet} \qquad \textsc{S-Binding}\ \frac{\pi\cdot\Gamma = \Delta}{\pi\cdot(\Gamma, x{:}_\rho\sigma) = \Delta, x{:}_{(\pi\cdot\rho)}\sigma}$$

$\boxed{\Gamma_1 + \Gamma_2}$ *(Context addition)*

$$\textsc{A-Binding}\ \frac{\Gamma_1 + \Gamma_2 = \Delta}{(\Gamma_1, x{:}_\pi\sigma) + (\Gamma_2, x{:}_\rho\sigma) = \Delta, x{:}_{(\pi+\rho)}\sigma} \qquad \textsc{A-Absent}\ \frac{\Gamma_1 + \Gamma_2 = \Delta \qquad x \notin \Gamma_2}{(\Gamma_1, x{:}_\pi\sigma) + \Gamma_2 = \Delta, x{:}_\pi\sigma} \qquad \textsc{A-Empty}\ \frac{}{\Gamma_1 + \bullet = \Gamma_1}$$

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

$$\textsc{E-Var}\ \frac{x{:}_1\forall\overline{a}.Q_1 \Rightarrow v \in \Gamma}{Q_1[\overline{\tau}/\overline{a}];\Gamma \vdash x : v[\overline{\tau}/\overline{a}]} \qquad \textsc{E-Abs}\ \frac{Q;\Gamma, x{:}_\pi\tau_1 \vdash e : \tau_2}{Q;\Gamma \vdash \lambda x.e : \tau_1 \to_\pi \tau_2}$$

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

$$\textsc{E-Pack}\ \frac{Q;\Gamma \vdash e : \tau[\overline{v}/\overline{a}]}{\omega\cdot Q \otimes Q_1[\overline{v}/\overline{a}];\Gamma \vdash \mathbf{pack}\ e : \exists\overline{a}.\tau \leftarrow Q_1}$$

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

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

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

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

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

Fig. 4. Qualified type system

- The type system has linear functions ($a \to_1 b$ in Figure 4). Despite our focus on linear constraints, we still need linearity in ordinary arguments. For example, the bind combinator for IO_L (Section 1) requires linear arrows: $(\gg\!=) :: IO_L\ a \multimap (a \multimap IO_L\ b) \multimap IO_L\ b$. Furthermore, the linearity of arrows interacts in interesting ways with linear constraints. Consider $f : a \to_\omega b$ and $x : \mathbf{q} \Rightarrow a$, then calling $f\ x$ would actually use $\mathbf{q}$ many times! That is, it must be impossible to derive $\mathbf{q};f{:}_\omega a \to_\omega b, x{:}_\omega\mathbf{q} \Rightarrow a \vdash f\ x : b$. Otherwise we could make, for instance, the *overusing* function from Section 2.3.3. You can check that $\mathbf{q};f{:}_\omega a \to_\omega b, x{:}_\omega\mathbf{q} \Rightarrow a \vdash f\ x : b$ indeed doesn't type check, because the scaling of Q_2 in rule E-App

ensures that the constraint would be $\omega \cdot \mathbf{q}$ instead. On the other hand, it's perfectly fine to have $\mathbf{q}; f :_\omega a \to_1 b, x :_\omega \mathbf{q} \multimap a \vdash f\ x : b$ when f is a linear function.
- 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). The inference algorithm of Section 5 will not make use of this possibility, but we revisit this capability in Section 8.1.
- Data types are not GADTs. This serves to considerably simplify the E-Case rule. It would be straightforward, yet tedious, to extend data types here to full GADTs.
- However, we include $\exists \overline{a}.\tau \circ\!\!= Q$, as introduced in Section 2.2, together with the **pack** and **unpack** constructions. These existentials act as roughly a GADT with a single constructor (see rules E-Pack and E-Unpack).
- A more minor difference with OutsideIn is that we have an explicit E-Sub rule, while OutsideIn uses simple constraint entailment directly in the relevant rules. In OutsideIn, only the E-Var rule needed subsumption; we would also need it for the E-Pack rule as well. So we preferred having one shared dedicated rule.

5 CONSTRAINT INFERENCE

Of course, the entire point of linear constraints is the promise that the compiler can handle these constraints for the programmer. Otherwise linear constraints are just plain Linear Haskell [Bernardy et al. 2017] with an extra set of symbols.

In this section, we are still following OutsideIn [Vytiniotis et al. 2011, Section 5]. We make, however, an important simplification: we infer constraints, but not types. That is in the syntax-directed constraint generation algorithm of Section 5.2, typing rules are still declarative rather than algorithmic. You can think of types as being inferred by an external oracle.

In OutsideIn – and in GHC – on the other hand, in order to support the wide range of features that modern Haskell boasts, the constraint solver is responsible not only for type class constraint resolution, but also for type inference in the more traditional sense. That is, the constraint solver deals with equality constraints and does unification. Doing so allows seamless integration of *e.g.* GADTs with their local equality assumptions and type families with their top-level equality axioms.

Not dealing with type inference greatly simplifies the presentation, as will be evident in the rest of the section. We can make this simplification for two reasons:

- Our system doesn't have equality constraints, and even when integrated into GHC, we do not allow equality constraints to be linear. This is because traditional unification algorithms are unsound for linear equalities as they will gladly reuse the same equality many times (or none at all). Linear equalities make sense (see *e.g.* [Shulman 2018], which puts linear equalities to great use), but they don't 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 can interfere with constraint resolution.

Our current focus is more narrow than a general typechecking algorithm for all of GHC's features. As a consequence of these two restrictions (no linear equalities and no multiplicity polymorphic constraints), type inference and (linear) class constraint resolution are completely orthogonal. Therefore, the syntax-directed constraint generation system presented in this section can legitimately assume that type inference is solved elsewhere, greatly simplifying the presentation.

5.1 Wanted constraints

Like OutsideIn [Vytiniotis et al. 2011, Figure 11, page 37], the constraints generated in Section 5.2 have a richer logical structure than the simple constraints. Let us follow GHC's terminology and

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

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

Fig. 5. Wanted-constraint entailment

call these *wanted constraints*: they are constraints which the constraint solver *wants* to prove.

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

The wanteds for linear constraints differ a little from the wanteds of OutsideIn. The first, superficial, difference is that our wanteds don't need existential quantification. This is one of the benefits that not doing type inference affords. In OutsideIn, existential quantifiation restricts which variables can be unified with which; but we are not doing any unification.

The more interesting difference is the presence of the extra $C_1 \,\&\, C_2$ construction (read C_1 *with* C_2). This is a connective from linear logic: a remarkable feature of linear logic is that every connective from classical or intuitionistic logic tends to appear in two flavours in linear logic. Where $C_1 \otimes C_2$ is the *multiplicative* conjunction, $C_1 \,\&\, C_2$ is the *additive* conjunction. Both connectives are conjunctions, but they differ rather dramatically in meaning: a proof of $C_1 \otimes C_2$ consumes both the (linear) assumptions consumed by a proof of C_1 and a proof of C_2. On the other hand in $C_1 \,\&\, C_2$, both the proof of C_1 and the proof of C_2 *must* consume the same assumptions, and that of $C_1 \,\&\, C_2$ will consume the same assumptions. The intuition, here, is that in $C_1 \,\&\, C_2$, only one of C_1 or C_2 will be eventually used. Much like if C_1 and C_2 were branches in a **case**-expression. Not coincidentally, $C_1 \,\&\, C_2$ will be used to generate constraints for **case**-expressions.

Finally, implication constraints differ from those of OutsideIn in two ways. First they are linear implications (an implication which must consume its assumption exactly once), where in OutsideIn they are intuitionistic implications. Additionally, there is a scaling factor π on implication. This scaling prescribes how many times the implication must be consumed, and is needed when extending scaling to all wanted constraints below. This scaling factor does not, however, change linear implication into intuitionistic implication. Concretely, $1 \cdot (\omega \cdot Q \multimap C)$ is an intuitionistic implication which must be consumed exactly once, while $\omega \cdot (1 \cdot Q \multimap C)$ is a linear implication which can be consumed many times.

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

Like in Section 4.2, we will typically drop the scaling factor for implication when it is 1 and write $Q \multimap C$ for $1 \cdot (Q \multimap C)$.

The meaning of wanted constraints is given by the inductive entailment relation $Q \vdash C$ given in Figure 5. This is a technical difference with OutsideIn where the meaning of wanteds is given directly by the constraint solver, whereas our solver (Section 5.3) has to be proved correct with respect to this entailment relation. We found that, in the case of linear logic, having this extra entailment relation made proofs much simpler.

Note that simple constraints always entail a wanted: we generate wanted constraints, but programs only mention simple constraints, and the left-hand side does come from the program. It is possible to relax the syntax of the left-hand side though, which GHC does [Bottu et al. 2017].

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

Lemma 5.1 (Inversion). *The inference rules of $Q \vdash C$ can be read bottom-up as well as top-down, as is required of $Q_1 \vdash Q_2$ in Definition 4.1. That is*

- *If $Q \vdash C_1 \otimes C_2$, then there exists Q_1 and Q_2 such that $Q_1 \vdash C_1$, $Q_2 \vdash C_2$, and $Q = Q_1 \otimes Q_2$.*
- *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 $Q = \pi \cdot Q_1$*

Proof. The cases $Q \vdash C_1 \,\&\, C_2$ and $Q \vdash \pi \cdot (Q_2 \multimap C)$ are immediate, since there is only one rule (C-With and C-Impl respectively) which can have them as their conclusion.

For $Q \vdash C_1 \otimes C_2$ we have two cases:

- either it is the conclusion of a C-Tensor rule, and the result is immediate.
- or it is the result of a C-Dom rule, in which case we have $C_1 = Q_1$, $C_2 = Q_2$, and the result follows from the definition of the entailment relation.

It may look like a slight change in the hypotheses would invalidate this proof. After all in a system with quantified constraints [Bottu et al. 2017], such as the current implementation of ghc, there are rules with non-atomic wanteds which do not introduce a connective. So, does this proof break when moving to the full ghc? Fortunately not!

What we really need is that any proof can be *rewritten* into a proof which ends in an introduction rule. Such proofs have been called *uniform* in Hodas and Miller [1994]. More precisely, a uniform proof is a proof where every rule whose conclusion has a non-atomic wanted is an introduction rule. The authors prove that all proofs in a fragment of linear logic can be rewritten into a uniform proof. This fragment includes quantified constraints and linear generalisations thereof. So, we can conclude that this lemma indeed extends to the complete set of constraints supported by ghc. □

Lemma 5.2 (Promotion). *If $Q \vdash C$, then $\pi \cdot Q \vdash \pi \cdot C$*

Lemma 5.3 (Inversion of promotion). *If $Q \vdash \pi \cdot C$ then $Q' \vdash C$ and $Q = \pi \cdot Q'$ for some Q'*

5.2 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 5.3. To generate constraints we introduce a judgement $\Gamma \vdash_{\mathsf{i}} e : \tau \rightsquigarrow C$ (defined in Figure 6). It is intended that C is understood as an output of this judgement. The definition $\Gamma \vdash_{\mathsf{i}} e : \tau \rightsquigarrow C$ is syntax directed, so it can directly be read as an algorithm, taking as input a term e (together with some type inference oracle, as discussed above) and returning a wanted constraint C.

The rules of Figure 6 are a mostly unsurprising translation of the rules of Figure 4 in the style of OutsideIn [Vytiniotis et al. 2011, Section 5.4]. In particular, the operations on typing contexts are the same as in Figure 4, and are not repeated here. Still, a few remarks are in order

- The most conspicuous difference is that, as we explained in Section 5.1, where OutsideIn uses a single kind of conjunctions, the E-Case rule needs two. OutsideIn accumulates constraints across branches, whereas we need to make sure that each branch of a **case**-expression consumes the same constraints.
  This is easily understood in terms of the file example of Section 1: if a file is closed in one branch of a **case**, it had better be closed in the other branches too. Otherwise, after the case its state will be unknown to the type system.

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

$$\textsc{G-Var}\ \frac{x{:}_1 \forall \overline{a}.Q \multimap v \in \Gamma}{\Gamma \vdash_{\mathsf{i}} x : v[\overline{\tau}/\overline{a}] \rightsquigarrow Q[\overline{\tau}/\overline{a}]} \qquad \textsc{G-Abs}\ \frac{\Gamma, x{:}_\pi \tau_0 \vdash_{\mathsf{i}} e : \tau \rightsquigarrow C}{\Gamma \vdash_{\mathsf{i}} \lambda x.e : \tau_0 \to_\pi \tau \rightsquigarrow C}$$

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

$$\textsc{G-Pack}\ \frac{\Gamma \vdash_{\mathsf{i}} e : \tau[\overline{v}/\overline{a}] \rightsquigarrow C}{\Gamma \vdash_{\mathsf{i}} \mathbf{pack}\ e : \exists \overline{a}.\tau \multimap Q \rightsquigarrow C \otimes Q[\overline{v}/\overline{a}]}$$

$$\textsc{G-Unpack}\ \frac{\Gamma_1 \vdash_{\mathsf{i}} e_1 : \exists \overline{a}.\tau_1 \multimap Q_1 \rightsquigarrow C_1 \qquad \overline{a}\ \text{fresh} \qquad \Gamma_2, x{:}_1 \tau_1 \vdash_{\mathsf{i}} e_2 : \tau \rightsquigarrow C_2}{\Gamma_1 + \Gamma_2 \vdash_{\mathsf{i}} \mathbf{unpack}\ x = e_1\ \mathbf{in}\ e_2 : \tau \rightsquigarrow C_1 \otimes (Q_1 \multimap C_2)}$$

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

$$\textsc{G-Let}\ \frac{\Gamma_1 \vdash_{\mathsf{i}} e_1 : \tau_1 \rightsquigarrow C_1 \qquad \Gamma_2, x{:}_\pi \tau_1 \vdash_{\mathsf{i}} e_2 : \tau \rightsquigarrow C_2}{\pi{\cdot}\Gamma_1 + \Gamma_2 \vdash_{\mathsf{i}} \mathbf{let}_\pi\ x = e_1\ \mathbf{in}\ e_2 : \tau \rightsquigarrow \pi{\cdot}C_1 \otimes C_2}$$

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

Fig. 6. Constraint generation

The reason this distinction is not made in an intuitionistic system like the original OutsideIn is elucidated by the fact that $\omega{\cdot}(C_1 \mathbin{\&} C_2) = \omega{\cdot}C_1 \otimes \omega{\cdot}C_2$. In other words, in a purely unrestricted context, linear constraints don't need two kinds of conjunctions either.

- Just like OutsideIn, the introduction of constraints local to a premise in the rules of Figure 4 corresponds to emitting an implicational constraint in Figure 6.
- However, the G-Let rule doesn't have an implicational constraint corresponding to the local constraint of rule E-Let: like in OutsideIn [Vytiniotis et al. 2011, Section 4.2], a **let** without a signature is not generalised. That being said, while not generalising is the easiest choice, here, it is not clear that it is the best in the presence of linear constraints, as we discuss in Section 8.1
- As we've observed throughout this section, not including type inference in the constraint generation algorithm significantly simplifies the presentation. There is no particular difficulty involved, however, in combining type inference and linear constraints; keeping in mind that equality constraints should never be linear.

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 4. That is, we are not defining a new type system but indeed an inference algorithm for the qualified type system.

Lemma 5.4 (Soundness of constraint generation). *For all Q_g if $\Gamma \vdash_{\mathsf{i}} e : \tau \rightsquigarrow C$ and $Q_g \vdash C$ then $Q_g; \Gamma \vdash e : \tau$*

Proof. Soundness is proved by induction on $\Gamma \vdash_{\mathsf{i}} e : \tau \rightsquigarrow C$, we only present the two most interesting cases, the full proof can be found in the supplementary material

E-App We have

- $\Gamma_1 + \pi \cdot \Gamma_2 \vdash_{\mathsf{i}} e_1\ e_2 : \tau \rightsquigarrow C_1 \otimes \pi \cdot C_2$
- $Q_g \vdash C_1 \otimes \pi \cdot C_2$
- $\Gamma_1 \vdash_{\mathsf{i}} e_1 : \tau_2 \to_\pi \tau \rightsquigarrow C_1$
- $\Gamma_2 \vdash_{\mathsf{i}} e_2 : \tau_2 \rightsquigarrow C_2$

By Lemma 5.1, there exist Q_1, Q_2 such that $Q_1 \vdash C_1$, $Q_2 \vdash C_2$, and $Q_g = Q_1 \otimes \pi \cdot Q_2$. Then, by induction hypothesis

- $Q_1; \Gamma_1 \vdash e_1 : \tau_2 \to_\pi \tau$
- $Q_2; \Gamma_2 \vdash e_2 : \tau_2$

Hence $Q_g; \Gamma_1 + \pi \cdot \Gamma_2 \vdash e_1\ e_2 : \tau$.

E-Unpack We have

- $\Gamma_1 + \Gamma_2 \vdash_{\mathsf{i}} \mathbf{unpack}\ x = e_1\ \mathbf{in}\ e_2 : \tau \rightsquigarrow C_1 \otimes Q' \multimap C_2$
- $Q_g \vdash C_1 \otimes Q' \multimap C_2$
- $\Gamma_1 \vdash_{\mathsf{i}} e_1 : \exists \overline{a}.\tau_1 \circ\!\!- Q' \rightsquigarrow C_1$
- $\Gamma_2, x :_\pi \tau_1 \vdash_{\mathsf{i}} e_2 : \tau \rightsquigarrow C_2$

By Lemma 5.1, there exist Q_1, Q_2 such that $Q_1 \vdash C_1$, $Q_2 \otimes Q' \vdash C_2$, and $Q_g = Q_1 \otimes Q_2$. Then, by induction hypothesis

- $Q_1; \Gamma_1 \vdash e_1 : \exists \overline{a}.\tau_1 \circ\!\!- Q'$
- $Q_2 \otimes Q; \Gamma_2 \vdash e_2 : \tau$

Therefore $Q_g; \Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\ x = e_1\ \mathbf{in}\ e_2 : \tau$.

□

5.3 Constraint solving

In this section, let us build a *constraint solver* whose purpose it is to prove that $Q_g \vdash C$ holds, as required by Lemma 5.4. The constraint solver is represented by the following judgement

$$\Phi; \Psi_i \vdash_{\mathsf{s}} C \rightsquigarrow \Psi_o$$

The judgement takes in two contexts: Φ, which holds all the unrestricted atomic constraints (scaled by ω) and Ψ_i, which holds all the linear atomic constraints. Due to associativity and commutativity of $\otimes$ we can always factor any simple constraint into such a pair of contexts. The unrestricted context Φ is an unordered set, while the multiplicative contexts Ψ_i and Ψ_o are ordered lists.

Linearity requires treating constraints as consumable resources. This is what Ψ_o is for: it is the list of the hypotheses of Ψ_i which haven't been consumed when proving Q_w. As suggested by the notation, it is an output of the algorithm.

There are three simplifications compared to OutsideIn's solver:

- As explained in Section 5.2, our solver does not need to do unification, so we do not return a substitution in the solver judgement.
- As discussed in Section 4.2, we omit top-level constraints, for the sake of simplification.
- In case the wanteds can not be solved completely, OutsideIn also returns a set of *residual* constraints: these are the remaining subproblems to solve. This set of residual constraints is simply quantified over in the type of top-level function definitions that don't have signatures. In our solver we omit these residual constraints, because we never infer a type scheme for a let without a signature (though we revisit this hypothesis in Section 8.1).
  It is important to reiterate this distinction: OutsideIn's residual constraints are constraints from the goal that remain to be solved, while our output constraints Ψ_o are constraints from the hypotheses that have yet to be used. OutsideIn's residuals are a tool for type inference, while our output constraints are an internal device for the solver.

$\boxed{\Phi; \Psi_i \vdash_{\mathsf{s}} C_w \rightsquigarrow \Psi_o}$ *(Constraint solving)*

$$\textsc{S-Simple}\ \frac{\Phi; \Psi_i \vdash_{\mathsf{s}}^{\mathsf{simp}} \pi{\cdot}\mathbf{q} \rightsquigarrow \Psi_o}{\Phi; \Psi_i \vdash_{\mathsf{s}} \pi{\cdot}\mathbf{q} \rightsquigarrow \Psi_o} \qquad \textsc{S-Mult}\ \frac{\Phi; \Psi_i \vdash_{\mathsf{s}} C_1 \rightsquigarrow \Psi_o' \quad \Phi; \Psi_o' \vdash_{\mathsf{s}} C_2 \rightsquigarrow \Psi_o}{\Phi; \Psi_i \vdash_{\mathsf{s}} C_1 \otimes C_2 \rightsquigarrow \Psi_o}$$

$$\textsc{S-ImplOne}\ \frac{\Phi, \omega{\cdot}Q_1; \Psi_i, Q_2 \vdash_{\mathsf{s}} C \rightsquigarrow \Psi_o \qquad \Psi_o \subseteq \Psi_i}{\Phi; \Psi_i \vdash_{\mathsf{s}} (\omega{\cdot}Q_1 \otimes Q_2) \multimap C \rightsquigarrow \Psi_o}$$

$$\textsc{S-Add}\ \frac{\Phi; \Psi_i \vdash_{\mathsf{s}} C_1 \rightsquigarrow \Psi_o \quad \Phi; \Psi_i \vdash_{\mathsf{s}} C_2 \rightsquigarrow \Psi_o}{\Phi; \Psi_i \vdash_{\mathsf{s}} C_1 \,\&\, C_2 \rightsquigarrow \Psi_o} \qquad \textsc{S-ImplMany}\ \frac{\Phi, \omega{\cdot}Q_1; Q_2 \vdash_{\mathsf{s}} C \rightsquigarrow \bullet}{\Phi; \Psi_i \vdash_{\mathsf{s}} \omega{\cdot}((\omega{\cdot}Q_1 \otimes Q_2) \multimap C) \rightsquigarrow \Psi_i}$$

Fig. 7. Constraint solver

Like OutsideIn [Vytiniotis et al. 2011, Section 5.5], the main solver infrastructure deals with solving the wanted constraints. To handle simple wanted constraints, we will need a domain-specific *simple-constraint solver* to be the algorithmic counterpart of the abstract entailment relation of Section 4.2. The main solver will appeal to this simple-constraint solver when solving atomic constraints. The simple-constraint solver is represented by the following judgement

$$\Phi; \Psi_i \vdash_{\mathsf{s}}^{\mathsf{simp}} \pi{\cdot}\mathbf{q} \rightsquigarrow \Psi_o$$

It has a similar structure to the main solver, but only deals with atomic constraints. Though the solver is parameterised in the particular choice of this simple-constraint solver, we will give an instantiation in Section 5.3.2.

Definition 5.5 (Simple-constraint solver). The simple constraint solver must verify that whenever $\Phi; \Psi_i \vdash_{\mathsf{s}}^{\mathsf{simp}} \pi{\cdot}\mathbf{q} \rightsquigarrow \Psi_o$, we have $\Psi_o \subseteq \Psi_i$ and $\Phi \otimes \Psi_i \Vdash \pi{\cdot}\mathbf{q} \otimes \Psi_o$ (where the Φ and the Ψ are interpreted as simple constraints by taking the tensor of their atomic constraints).

5.3.1 The solver. To extend the solver to all wanted constraints, we use a linear proof search algorithm based on the recipe given by Cervesato et al. [2000]. Figure 7 presents the rules of the constraint solver.

- The S-Mult rule proceeds by solving one side of a conjunction first, then passing the output constraint to the other side. The unrestricted context is shared between both sides.
- The S-Add rule handles additive conjunction. Here, the linear constraints are also shared between the branches (since additive conjunction is generated from case expressions, only one of them is actually going to be executed). Note that both branches must consume exactly the same resources.
- Implications are handled by S-ImplOne and S-ImplMany, for solving linear and unrestricted wanted implications respectively. In both cases, the assumption of the implication is split to the unrestricted $\omega{\cdot}Q_1$ and the linear Q_2 parts (this can be done deterministically). When solving a linear implication, we add the assumptions to their respective context, and proceed with solving the conclusion. Importantly (see Section 5.3.2 below), the linear assumptions are added to the front of the list. The side condition requires that the output context is a subset of the input context: this is to ensure that the implication actually consumes its assumption and doesn't leak it to the ambient context. Solving unrestricted implications only allows the conclusion of the implication to be solved using its own linear assumption, but none of the other linear constraints. This is because unrestricted implications only use their own assumption linearly, but use everything from the ambient context ω times.

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

$$\text{Q-Hyp}\ \frac{}{\omega{\cdot}Q \otimes \pi{\cdot}\mathbf{q} \Vdash \pi{\cdot}\mathbf{q}} \qquad \text{Q-Prod}\ \frac{Q_1 \Vdash Q_1' \quad Q_2 \Vdash Q_2'}{Q_1 \otimes Q_2 \Vdash Q_1' \otimes Q_2'} \qquad \text{Q-Empty}\ \frac{}{\omega{\cdot}Q \Vdash \varepsilon}$$

(a) Entailment relation

$\boxed{\Phi; \Psi \vdash_{\mathsf{s}}^{\mathsf{simp}} \pi{\cdot}\mathbf{q} \rightsquigarrow \Psi_o}$ *(Simple-constraint solver)*

$$\text{Simp-AtomMany}\ \frac{}{\Phi, \omega{\cdot}\mathbf{q}; \Psi \vdash_{\mathsf{s}}^{\mathsf{simp}} \omega{\cdot}\mathbf{q} \rightsquigarrow \Psi} \qquad \text{Simp-AtomOneL}\ \frac{\mathbf{q} \notin \Psi_2 \qquad \omega{\cdot}\mathbf{q} \notin \Phi}{\Phi; \Psi_1, \mathbf{q}, \Psi_2 \vdash_{\mathsf{s}}^{\mathsf{simp}} \mathbf{q} \rightsquigarrow \Psi_1, \Psi_2} \qquad \text{Simp-AtomOneU}\ \frac{\mathbf{q} \notin \Psi}{\Phi, \omega{\cdot}\mathbf{q}; \Psi \vdash_{\mathsf{s}}^{\mathsf{simp}} \mathbf{q} \rightsquigarrow \Psi}$$

(b) Simple-constraint solver

Fig. 8. A stripped-down constraint domain

5.3.2 A simple-constraint solver. So far, the simple-constraint domain has been an abstract parameter. In this section, though, let us give a concrete domain which supports our examples.

For the sake of our examples, we really need very little: linear constraints are simply names as far as the type checker is concerned. So it is enough for the entailment relation (Figure 8a) to prove **q** if and only if it is already on the left hand side, with some restriction for linearity of course.

The corresponding simple-constraint solver (Figure 8b) is more interesting. The first thing to notice is that the rules are deterministic: in any circumstances, only one of the three rules can apply. This is a central tenet of OutsideIn [Vytiniotis et al. 2011, Section 6.4]: it never makes guesses.

Figure 8b is also where the fact that the Ψ are lists comes into play. Indeed, rule Simp-AtomOneL takes great care to use the most recent occurrence of **q** (remember that rule S-ImplOne adds the new hypotheses on the top of the list). To understand why, consider the following example:

```
f :: FilePath → IO_L ()
f fp = do {pack h ← openFile fp
          ; let {cl :: Open h ⊸ IO_L ()
                ; cl = closeFile h}
          ; cl}
```

In this example it is quite clear, that the programmer meant for *closeFile* to use *Open h* introduced locally in *cl*. In fact this is the only possible way to attribute constraints.

Another interesting feature of the solver (Figure 8b) is that no rule solves a linear constraint if it appears both in the unrestricted and the linear context. Consider the following (contrived) API:

```
class C
giveC :: (C ⇒ Int) → Int
useC :: C ⊸ Int
```

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

```
bad :: C ⊸ (Int, Int)
bad = (giveC useC, useC)
```

$$
\begin{array}{lcll}
\sigma & ::= & \forall \overline{a}.\tau & \text{Type schemes} \\
\tau, \upsilon & ::= & \ldots \mid \exists \overline{a}.\tau \circ\!\!- \upsilon & \text{Types} \\
e & ::= & \ldots \mid \mathbf{pack}\,(e_1, e_2) \mid \mathbf{unpack}\,(y, x) = e_1\ \mathbf{in}\ e_2 & \text{Expressions}
\end{array}
$$

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

$$
\textsc{L-Pack}\ \frac{\Gamma_1 \vdash e_1 : \tau_1[\overline{\upsilon}/\overline{a}] \qquad \Gamma_2 \vdash e_2 : \tau_2[\overline{\upsilon}/\overline{a}]}{\Gamma_1 + \Gamma_2 \vdash \mathbf{pack}\,(e_1, e_2) : \exists \overline{a}.\tau_2 \circ\!\!- \tau_1}
$$

$$
\textsc{L-Unpack}\ \frac{\Gamma_1 \vdash e_1 : \exists \overline{a}.\tau_2 \circ\!\!- \tau_1 \qquad \overline{a}\ \text{fresh} \qquad \Gamma_2, y :_1 \tau_1, x :_1 \tau_2 \vdash e_2 : \tau}{\Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\,(y, x) = e_1\ \mathbf{in}\ e_2 : \tau}
$$

Fig. 9. Core calculus (subset)

It is possible to give a type derivation to *bad* in the qualified type system of Section 4. In fact, the constraint assignment is unambiguous: the left-most *useC* must use the unrestricted C, while the right-most must use the linear C. This assignment, however, would require of the constraint solver to make a guess when solving the left-most *useC*. So in the spirit of OutsideIn, *bad* is rejected.

6 DESUGARING

The semantics of our language is given by desugaring it into a simpler core language: a mild 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.

6.1 The core calculus

The core calculus is a variant of the type system defined in Section 4 without implicit arguments. Figure 9 highlights the differences from the qualified system:

- Type schemes σ do not support qualified types.
- Existentially quantified types $\exists \overline{a}.\tau_2 \circ\!\!- \tau_1$ now represent a (linear) pair of values. Accordingly, **pack** and **unpack** operate on pairs.

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

- We don't have multiplicity polymorphism.
- We need, on the other hand, type polymorphism.
- Polymorphism is implicit rather than explicit. This is not an essential difference but it simplifies the presentation.
- We have existential types. These can be realised in regular Haskell as a family of GADTs.

In addition, we shall assume the existence of 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 $\mathbf{Ur} : \forall a.a \to_\omega \mathbf{Ur}\ a$

These are regular data types and constructors of the language, they are consumed with **case**.

6.2 Inferred constraints

Using Lemma 5.2 together with Definition 5.5 we know that if $\Gamma \vdash_{\mathsf{i}} e : \tau \rightsquigarrow C$ and $\Phi; \Psi \vdash_{\mathsf{s}} C \rightsquigarrow \bullet$, then $\Phi \otimes \Psi; \Gamma \vdash e : \tau$. It only remains to desugar derivations of $Q; \Gamma \vdash e : \tau$ into the core calculus.

$$\begin{cases} \mathbf{Ev}\,(1\cdot\mathbf{q}) & = & \mathbf{Ev}\,(\mathbf{q}) \\ \mathbf{Ev}\,(\omega\cdot\mathbf{q}) & = & \mathbf{Ur}\,(\mathbf{Ev}\,(\mathbf{q})) \\ \mathbf{Ev}\,(\varepsilon) & = & \mathbf{1} \\ \mathbf{Ev}\,(Q_1 \otimes Q_2) & = & \mathbf{Ev}\,(Q_1) \otimes \mathbf{Ev}\,(Q_2) \end{cases}$$

(a) Evidence passing

$$\begin{cases} \mathbf{Ds}\,(z; Q; \Gamma \vdash x : v[\overline{\tau}/\overline{a}]) = x\ z \\ \mathbf{Ds}\,(z; Q_1 \otimes Q_2; \Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\ x = e_1\ \mathbf{in}\ e_2 : t) = \\ \quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, z_2) \to \\ \qquad \mathbf{unpack}\ (z', x) = \mathbf{Ds}\,(z'; Q_1; \Gamma_1 \vdash e_1 : \exists\ \overline{a}.\tau_1 \multimap Q)\ \mathbf{in} \\ \qquad \mathbf{let}_1\ z_2{}' = (z_2, z')\ \mathbf{in} \\ \qquad \mathbf{Ds}\,(z_2{}'; Q_2 \otimes Q; \Gamma_2, x :_1 \tau_1 \vdash e_2 : t)\} \\ \mathbf{Ds}\,(z; Q; \Gamma \vdash e : t) = \\ \quad \mathbf{let}_1\ z' = \mathbf{Ev}\,(Q \Vdash Q_1)\ z\ \mathbf{in}\ \mathbf{Ds}\,(z'; Q_1; \Gamma \vdash e : t) \\ \ldots \end{cases}$$

(b) Desugaring (subset)

Fig. 10. Evidence passing and desugaring

6.3 From qualified to core

6.3.1 Evidence. In order to desugar derivations of the qualified system to the core calculus, we will use the classic technique known as evidence-passing style[4].

To do so, we shall require some more material from constraints. Namely, we will assume a type $\mathbf{Ev}\,(\mathbf{q})$ for each atomic constraint, defined in Figure 10a. It is extended to all simple constraints. It ought to be noted that $\mathbf{Ev}\,(Q)$ is not technically well defined, as the rules in Section 4.2 define the syntax as being quotiented by associativity and commutativity of the tensor product, and idempotence of unrestricted constraints. But core language data types (or Haskell's for that matter) are not so quotiented. It's not actually hard to fix the imprecision: give a name to each atomic constraint, and operate on canonical representatives of the equivalence classes. This is actually essentially how GHC deals with constraints today. It is also the mechanism that our prototype implementation (see Section 7) uses. However, we have preferred keeping this section a little imprecise, in order to save the rest of the article from the non-trivial extra tedium that the more precise presentation entails.

Furthermore we shall require that for every Q_1 and Q_2 such that $Q_1 \Vdash Q_2$, there is a (linear) function $\mathbf{Ev}\,(Q_1 \Vdash Q_2) : \mathbf{Ev}\,(Q_1) \to_1 \mathbf{Ev}\,(Q_2)$.

Let us now define a family of functions $\mathbf{Ds}\,(\_)$ to translate the type schemes, types, and typing derivations of the qualified system into the types, type schemes, and terms of the core calculus.

6.3.2 Translating types. Type schemes σ are translated by turning the implicit argument Q into an explicit one of type $\mathbf{Ev}\,(Q)$. Translating types τ proceeds as expected.

$$\left\{ \mathbf{Ds}\,(\forall\overline{a}.Q \multimap \tau) \quad = \quad \forall\overline{a}.\mathbf{Ev}\,(Q) \to_1 \mathbf{Ds}\,(\tau) \right. \qquad \begin{cases} \mathbf{Ds}\,(\tau_1 \to_\pi \tau_2) & = & \mathbf{Ds}\,(\tau_1) \to_\pi \mathbf{Ds}\,(\tau_2) \\ \mathbf{Ds}\,(\exists\overline{a}.\tau \circ\!\!- Q) & = & \exists\overline{a}.\mathbf{Ds}\,(\tau) \circ\!\!- \mathbf{Ev}\,(Q) \end{cases}$$

6.3.3 Translating terms. Let us finally build, given a derivation $Q; \Gamma \vdash e : \tau$, an expression $\mathbf{Ds}\,(z; Q; \Gamma \vdash e : \tau)$, such that $\Gamma, z :_1 \mathbf{Ev}\,(Q) \vdash \mathbf{Ds}\,(z; Q; \Gamma \vdash e : \tau) : \mathbf{Ds}\,(\tau)$ (for some variable z). Even though we abbreviate the derivation as its conclusion, 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 10b.

The cases correspond to the E-Var, E-Unpack, 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 **unpack**

[4]This technique is also often called dictionary-passing style because, in the case of type classes, evidences are dictionaries, and because type classes were the original form of constraints in Haskell.

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 $z : \mathbf{Ev}\,(Q)$ then proceeds to desugar e with the resulting evidence for Q_1. Crucially, since $\mathbf{Ds}\,(\_)$ 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, indeed, $\Gamma, z{:}_1\mathbf{Ev}\,(Q) \vdash \mathbf{Ds}\,(z; Q; \Gamma \vdash e : \tau) : \mathbf{Ds}\,(\tau)$ as announced.

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.

7 IMPLEMENTATION

We have written a prototype implementation 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 – a fairly local change.

The constraint solver maintains a set of givens (the *inert* set), which corresponds to the Φ and Ψ contexts in our solver judgements in Section 5.3. A property of the inert set is that constraints contained do not interact pairwise. These interactions are dictated by the constraint domain. For example, equality constraints interact with other constraints by applying a substitution.

The treatment of implication constraints is of particular interest. Implications, by their nature, introduce assumptions which do not necessarily hold in the outer context, therefore recording these assumptions in the inert set is a destructive operation. To ensure proper scoping, GHC creates a fresh copy of the inert set for each nested implication it solves, so these destructive operations do not leak out. We modify the inert set so that for each constraint stored in it, the level (or depth) of the implication is recorded alongside it. Each interaction with nested assumptions (which might give rise to additional derived givens) is recorded at the appropriate level and multiplicity (decomposing a constraint tuple into its constituent parts is done by the simplifier, which must now record the multiplicities of the components).

Atomic wanteds are then solved by finding a matching given in the inert set (or a top-level given, which is not relevant to linear constraints). When a linear given is used to solve a linear wanted, our prototype removes the given from the inert set so it can not be used again. Before, the inert set only held a single copy of each given, but now it must hold multiple copies of linear givens, together with their implication level. When solving an atomic wanted, the matching given with the largest implication level (i.e. the innermost given) is selected, as per the Simp-AtomOneL rule.

When an implication is finally solved, we must check that every linear constraint introduced in this implication was consumed, which is done by checking that the level of every linear constraint in the inert set is less than the implication.

When a linear equality constraint is encountered, it is automatically promoted to an unrestricted one and handled accordingly. This may happen in many different scenarios, as the entailment relation of GHC's constraint domain does produce many equalities, thus we need to ensure they are turned into unrestricted before interacting with the inert set. GHC Core already represents the equality constraint as a boxed type, so we can simply modify it to store an unrestricted payload.

As constraint solving proceeds, the compiler pipeline constructs a term in an intrinsically typed language known as GHC Core [Sulzmann et al. 2007]. In Core, type class constraints are turned into explicit evidence (see Section 6). Thanks to being fully annotated, Core has decidable typechecking which is useful in sanity checking modifications to the compiler. Thus, the soundness of our implementation is verified by the Core typechecker, which already supports linearity.

8 EXTENSIONS

The system presented in Section 4 and Section 5 is already capable of supporting the examples in Section 2 and Section 3. In this section, we consider some potential avenues for extensions.

8.1 let generalisation

As discussed in Section 5.2, the G-LET rule of our constraint generator does not generalise the type of let bindings. Not doing let generalisation is in line with GHC's existing behaviour [Vytiniotis et al. 2011, Section 4.2]. There, this behaviour was guided by concerns around inferring type variables, which is harder in the presence of local equality assumptions (that is, GADT pattern matching).

In this section, however, we argue that generalising over linear constraints may, in fact, improve user experience. Let us revisit the *firstLine* example from Section 1, but this time, instead of executing *closeFile* directly, we assign it to a variable in a let binding:

$$
\begin{aligned}
&\mathit{firstLine} :: \mathit{FilePath} \to \mathit{IO}_L\ \mathit{String}\\
&\mathit{firstLine}\ \mathit{fp} = \mathbf{do}\ \{\mathbf{pack}!\ h \leftarrow \mathit{openFile}\ \mathit{fp}\\
&\qquad\qquad ;\ \mathbf{let}\ \mathit{closer} = \mathit{closeFile}\ h\\
&\qquad\qquad ;\ \mathbf{pack}!\ \mathit{xs} \leftarrow \mathit{readLine}\ h\\
&\qquad\qquad ;\ \mathit{closer}\\
&\qquad\qquad ;\ \mathit{return}\ \mathit{xs}\}
\end{aligned}
$$

This program looks reasonable; however, it is rejected. The type of *closer* is $\mathit{IO}_L\ ()$, which means that the definition of *closer* consumes the linear constraint $\mathit{Open}\ h$. So, by the time we try to *readLine h*, the constraint is no longer available.

What the programmer really meant, here, was for *closer* to have type $\mathit{Open}\ h \multimap \mathit{IO}_L\ ()$. After all a **let** definition is not part of the sequence of instructions, it is just a definition for later; it is not supposed to consume the current state of the file. With no let generalisation, the only way to give *closer* the type $\mathit{Open}\ h \multimap \mathit{IO}_L\ ()$ is to give *closer* a type signature. In current GHC, we can't write that signature down, since the type variable h is not bound in the program text, and there is no syntax for binding it (though see Eisenberg et al. [2018] for a proposed syntax). But even ignoring this, it would be rather unfortunate that the default behaviour of **let**, in presence of linear constraints, almost never be what the programmer wants.

To handle **let**-generalisation, let us consider the following rule

$$
\textsc{G-LetGen}\quad
\frac{\begin{array}{c}\Gamma_1 \vdash_{\mathsf{i}} e_1 : \tau_1 \rightsquigarrow C_1\\ Q_r \otimes Q \vdash C_1\\ \Gamma_2, x :_\pi Q \multimap \tau_1 \vdash_{\mathsf{i}} e_2 : \tau \rightsquigarrow C_2\end{array}}{\pi \cdot \Gamma_1 + \Gamma_2 \vdash_{\mathsf{i}} \mathbf{let}_\pi\ x = e_1\ \mathbf{in}\ e_2 : \tau \rightsquigarrow \pi \cdot Q_r \otimes C_2}
$$

This rule is non-deterministic, as it requires finding Q_r and Q. We can modify the constraint solver of Section 5.3 to find $Q_r \otimes Q$: it's the residual constraint, from OutsideIn, that we have omitted. But we still have to determine how to split the residual into Q_r and Q. What we would like to say is "Q is the set of linear constraints". But it's not clear how to make it formal.

Any predictable strategy would do: as long as it's an instance of the G-LetGen rule, constraint generation will be sound. Experience will tell whether we can find a better suited strategy than the current one, which never generalises any constraint.

8.2 Empty cases

Throughout the article we have assumed that **case**-expressions always have a non-empty list of alternatives. This is, incidentally, also how Haskell originally behaved; though GHC now has an *EmptyCase* extension to allow empty lists of alternatives.

Not allowing empty lists of alternatives is, therefore not terrible in principle. Though it makes empty types more awkward than they need to be, and, of course, to support the entirety of GHC, we will need to support empty lists of alternatives.

The reason why it has been omitted from the rest of the article is that generating constraints for an empty requires an 0-ary version of $C_1 \,\&\, C_2$, usually written $\top$ in Linear Logic. The corresponding entailment rule would be

$$\text{C-Top}\quad \frac{}{Q \vdash \top}$$

That is $\top$ is unconditionally true, and can consume any number of linear given constraints. The C-Top rule thus induces a considerable amount of non-determinism. Eliminating the non-determinism induced by $\top$ is ultimately what Cervesato et al. [2000] builds up to. Their methods can be adapted to the constraint solver of Section 5.3 without any technical difficulty. We chose, however, to keep empty cases out the presentation because has a very high overhead and would distract from the point. Instead, we refer readers to Cervesato et al. [2000, Section 4] for a careful treatment of $\top$.

9 RELATED WORK

Rust. The memory ownership example of Section 3 is strongly inspired by Rust. There seems to be something inescapable about Rust's model (itself inspired by C++'s move semantics) of owned and borrowed references, as the API in Section 3 follows naturally from the requirements (namely a pure interface for mutable arrays with the ability to freeze arrays). It is not clear that any other API could be essentially different.

Rust is built with ownership and borrowing for memory management from the ground up. As a consequence, it has a much more convenient syntax than Linear Haskell with linear constraints can propose. Rust's convenient syntax comes at the price that it is almost impossible to write tail-recursive functions, which is surprising from the perspective of a functional programmer.

On the other hand, the focus of Linear Haskell as well as linear constraints is to provide programmers with the tools to create safe interfaces and libraries. The language itself is agnostic about what linear constraints mean. Although linear constraints don't have the convenience of Rust's syntax, we expect that they will support a greater variety of abstractions. It is worth noting, though, that Rust programmers have come up with varied abstractions which leverage borrowing, going beyond memory management (for instance, safe file handling).

Section 3 shows how to make a Rust-like DSL in Haskell with linear constraints. It is not clear that it is possible at all to make a Linear-Haskell-like DSL in Rust.

Languages with capabilities. 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, and ATS is the closest to our system with an explicit appeal to linear logic, and the capabilities (known as *stateful views*) being a part of a bigger language. However, ATS does not have full inference of capabilities.

Other than that, both systems have a lot of similarities. They have a finer-grained capability system than is expressible in Rust (or our Section 3) which makes it possible to change the type of a reference cell upon write. They also eschew scoped borrowing in favour of more traditional read and write capabilities. In exchange, neither Mezzo nor ATS support $O(1)$ freezing like in Section 3.

Mezzo, being geared towards functional programming, does support freezing, but freezing a nested data structure requires traversing it. As far as we know, ATS doesn't support freezing. ATS is more oriented towards system programming.

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.

Though both Mezzo and ATS have an advantage that we don't: they assume that their instructions are properly sequenced, whereas basing linear constraints on Haskell, a lazy language, we are forced to make sequencing explicit in APIs.

Logic programming. There are a lot of commonalities between GHC's constraint and logic programs. Traditional type classes can be seen as Horn Clauses programs, much like Prolog's programs. GHC puts further restrictionss 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 programs. A generalisation of the Hereditary Harrop fragment to linear logic, described by Hodas and Miller [1994], is the foundation of the Lolli language. Together with this fragment, the authors coin the notion of *uniform* proof. A fragment where uniform proofs are complete supports goal-oriented proof search, like Prolog does.

Uniformity is used in the proof of Lemma 5.1, which, in turn is used in the proof of the soundness lemma 5.4. This seems to indicate that goal-oriented search is baked into the definition of OutsideIn. An immediate consequence of this observation, however, is that the fact that that the fragment of linear logic described by Hodas and Miller [1994] (and for which Cervesato et al. [2000] provides a refined search strategy) contains the Hereditary Harrop fragment of intuitionistic logic guarantees that quantified constraints don't break our proofs.

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.

We also demonstrate how an existing constraint solver can be extended to handle linearity. Our design of linear constraints fits nicely into Haskell. Indeed, linear constraints can be thought of as an extension of Haskell's type class mechanism. This way, the design also integrates well into GHC, as demonstrated by our prototype implementation, which required modest changes to the compiler. Remarkably, all we needed was to adapt Cervesato et al. [2000] to the OutsideIn framework. It is also quite serendipitous that the notion of uniform proof from Hodas and Miller [1994], which was introduced to prove the completeness of a proof search strategy, ends up being crucial to the soundness of constraint generation.

In some cases, like the file example of Section 1, linear constraints are a mere convenience that reduce line noise and make code more idiomatic. But the memory management API of Section 3 is not feasible without linear constraints. Certainly, ownership proofs could be managed manually, but it is hard to imagine a circumstance where this tedious task would be worth the cost.

This, really, is the philosophy of linear constraints: lowering the cost of linear types so that more theoretical applications become practical applications. And we achieved this at a surprisingly low price: teaching linear logic to GHC's constraint solver.

REFERENCES

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

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, Joachim Breitner, and Simon Peyton Jones. 2018. Type Variables in Patterns. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/3242744.3242753

Dan R. Ghica and Alex I. Smith. 2014. Bounded Linear Types in a Resource Semiring. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*. 331–350. https://doi.org/10.1007/978-3-642-54833-8_18

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

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/10.1006/inco.1994.1036

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

Oleg Kiselyov and Chung-chieh Shan. 2004. Functional Pearl: Implicit Configurations–or, Type Classes Reflect the Values of Types. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell* (Snowbird, Utah, USA) *(Haskell '04)*. Association for Computing Machinery, New York, NY, USA, 33–44. https://doi.org/10.1145/1017472.1017481

John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. Association for Computing Machinery, New York, NY, USA, 24–35. https://doi.org/10.1145/178243.178246

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

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

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

Niki Vazou, Patrick M. Rondon, and Ranjit Jhala. 2013. Abstract Refinement Types. In *Programming Languages and Systems*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 209–228.

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

Hongwei Xi. 2017. Applied Type System: An Approach to Practical Programming with Theorem-Proving. *CoRR* abs/1703.08683 (2017). arXiv:1703.08683 http://arxiv.org/abs/1703.08683

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.

$$
\begin{array}{llll}
a, b & ::= & \ldots & \text{Type variables} \\
x, y & ::= & \ldots & \text{Expression variables} \\
K & ::= & \ldots & \text{Data constructors} \\
\sigma & ::= & \forall \overline{a}.\tau & \text{Type schemes} \\
\tau, \upsilon & ::= & a \mid \exists \overline{a}.\tau \multimap \upsilon \mid \tau_1 \to_\pi \tau_2 \mid T\,\overline{\tau} & \text{Types} \\
\Gamma, \Delta & ::= & \bullet \mid \Gamma, x{:}_\pi \sigma & \text{Contexts} \\
e & ::= & x \mid K \mid \lambda x.e \mid e_1\ e_2 \mid \mathbf{pack}\ (e_1, e_2) & \text{Expressions} \\
 & \mid & \mathbf{unpack}\ (y, x) = e_1\ \mathbf{in}\ e_2 \mid \mathbf{case}_\pi\ e\ \mathbf{of}\ \{\overline{K_i\ \overline{x_i} \to e_i}\} & \\
 & \mid & \mathbf{let}_\pi\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{let}_\pi\ x : \sigma = e_1\ \mathbf{in}\ e_2 &
\end{array}
$$

Fig. 11. Grammar of the core calculus

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

$$\text{L-Var}\ \frac{x{:}_1 \forall \overline{a}.\upsilon \in \Gamma}{\Gamma \vdash x : \upsilon[\overline{\tau}/\overline{a}]} \qquad \text{L-Abs}\ \frac{\Gamma, x{:}_\pi \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to_\pi \tau_2} \qquad \text{L-App}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \to_\pi \tau \quad \Gamma_2 \vdash e_1 : \tau_1}{\Gamma_1 + \pi{\cdot}\Gamma_2 \vdash e_1\ e_2 : \tau} \qquad \text{L-Pack}\ \frac{\Gamma_1 \vdash e_1 : \tau_1[\overline{\upsilon}/\overline{a}] \quad \Gamma_2 \vdash e_2 : \tau_2[\overline{\upsilon}/\overline{a}]}{\Gamma_1 + \Gamma_2 \vdash \mathbf{pack}\ (e_1, e_2) : \exists \overline{a}.\tau_2 \multimap \tau_1}$$

$$\text{L-Unpack}\ \frac{\Gamma_1 \vdash e_1 : \exists \overline{a}.\tau_2 \multimap \tau_1 \quad \overline{a}\ \text{fresh} \quad \Gamma_2, y{:}_1 \tau_1, x{:}_1 \tau_2 \vdash e_2 : \tau}{\Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\ (y, x) = e_1\ \mathbf{in}\ e_2 : \tau} \qquad \text{L-Let}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \quad \Gamma_2, x{:}_\pi \sigma \vdash e_2 : \tau}{\pi{\cdot}\Gamma_1 + \Gamma_2 \vdash \mathbf{let}_\pi\ x : \sigma = e_1\ \mathbf{in}\ e_2 : \tau}$$

$$\text{L-Case}\ \frac{\Gamma_1 \vdash e : T\,\overline{\tau} \quad K_i : \forall \overline{a}.\overline{\upsilon_i} \to_{\overline{\pi_i}} T\,\overline{a} \quad \Gamma_2, x_i{:}_{(\pi \cdot \pi_i)} \upsilon_i[\overline{\tau}/\overline{a}] \vdash e_i : \tau}{\pi{\cdot}\Gamma_1 + \Gamma_2 \vdash \mathbf{case}_\pi\ e\ \mathbf{of}\ \{\overline{K_i\ \overline{x_i} \to e_i}\} : \tau}$$

Fig. 12. Core calculus type system

A FULL DESCRIPTIONS

In this appendix, we give, for reference, complete descriptions of the type systems, functions, etc… that we have abbreviated in the main body of the article.

A.1 Core calculus

This is the complete version of the core calculus described in Section 6.1. The full grammar is given by Figure 11 and the type system by Figure 12.

A.2 Desugaring

The complete definition of the desugaring function from Section 6 can be found in Figure 13.

For the sake of concision, we allow ourselves to write nested patterns in case expressions of the core language. Desugaring nested patterns into atomic case expression is routine.

In the complete description, we use a device which was omitted in the main body of the article. Namely, we'll need a way to turn every $\mathbf{Ev}\ (\omega{\cdot}Q)$ into an $\mathbf{Ur}\ (\mathbf{Ev}\ (Q))$. For any $e : \mathbf{Ev}\ (\omega{\cdot}Q)$, we shall write $\underline{e}_Q : \mathbf{Ur}\ (\mathbf{Ev}\ (\omega{\cdot}Q))$. As a shorthand, particularly useful in nested patterns, we will write

$\mathbf{case}_\pi\ e\ \mathbf{of}\ \{\underline{x}_Q \to e'\}$ for $\mathbf{case}_\pi\ \underline{e}_Q\ \mathbf{of}\ \{\mathbf{Ur}\ x \to e'\}$.

$$\begin{cases} \underline{e}_\varepsilon & = & \mathbf{case}_1\ e\ \mathbf{of}\ \{() \to \mathbf{Ur}\ ()\} \\ \underline{e}_{1\cdot\mathbf{q}} & = & e \\ \underline{e}_{\omega\cdot\mathbf{q}} & = & \mathbf{case}_1\ e\ \mathbf{of}\ \{\mathbf{Ur}\ x \to \mathbf{Ur}\ (\mathbf{Ur}\ x)\} \\ \underline{e}_{Q_1\otimes Q_2} & = & \mathbf{case}_1\ e\ \mathbf{of}\ \{(\underline{x}_{Q_1}, \underline{y}_{Q_2}) \to \mathbf{Ur}\ (x, y)\} \end{cases}$$

We will omit the Q in $\underline{e}_Q$ and write $\underline{e}$ when it can be easily inferred from the context.

B PROOFS OF THE LEMMAS

B.4 Lemmas on the qualified type system

Proof of Lemma 4.2. Let us prove this lemma by induction on the syntax of Q_2:

- If $Q_2 = \rho\cdot\mathbf{q}$, then $\pi\cdot Q_1 \Vdash (\pi\cdot\rho)\cdot\mathbf{q}$ holds by Definition 4.1.
- If $Q_2 = Q_2' \otimes Q_2''$, then, by Definition 4.1, we know that $Q_1 = Q_1' \otimes Q_1''$ for some Q_1' and Q_1'', and that $Q_1' \Vdash Q_2'$ and $Q_1'' \Vdash Q_2''$. By induction hypothesis, we have $\pi\cdot Q_1' \Vdash \pi\cdot Q_2'$ and $\pi\cdot Q_1'' \Vdash \pi\cdot Q_2''$. From which it follows that $\pi\cdot Q_1 \Vdash \pi\cdot Q_2$.
- If $Q_2 = \varepsilon$, then, by Definition 4.1, we know that $Q_1 = \omega\cdot Q_1'$, for some Q_1' and the result follows from Lemma B.1, since $\pi\cdot Q_1 = \pi\cdot(\omega\cdot Q_1') = \omega\cdot Q_1'$.

□

Proof of Lemma 4.3. Let us prove this lemma by induction on the syntax of Q_2

- If $Q_2 = \rho\cdot\mathbf{q}$, then, by Definition 4.1, there exists Q_1' such that $Q_1 = \pi\cdot Q_1'$ and $Q_1' \Vdash \rho\cdot\mathbf{q}$.
- If $Q_2 = Q_2' \otimes Q_2''$, then, by Definition 4.1, we know that $Q_1 = Q_1' \otimes Q_1''$ for some Q_1' and Q_1'', and that $Q_1' \Vdash \pi\cdot Q_2'$ and $Q_1'' \Vdash \pi\cdot Q_2''$ (remember that, by definition, $\pi\cdot Q_2 = \pi\cdot Q_2' \otimes \pi\cdot Q_2''$). By induction hypothesis, we have constraints Q' and Q'', such that $Q_1' = \pi\cdot Q'$ and $Q_1'' = \pi\cdot Q''$, and $Q' \Vdash Q_2'$ and $Q'' \Vdash Q_2''$. It follows that $Q_1 = \pi\cdot(Q' \otimes Q'')$ and $Q' \otimes Q'' \Vdash Q_2$.
- If $Q_2 = \varepsilon$, then, since $\pi\cdot\varepsilon = \varepsilon$, by Definition 4.1, $Q_1 = \varepsilon$ and the result is immediate.

□

Lemma B.1. *The following equality holds* $\pi\cdot(\rho\cdot Q) = (\pi\cdot\rho)\cdot Q$

Proof. This result can be proved by a straightforward induction on the structure of Q. □

B.5 Lemmas on constraint inference

Proof of Lemma 5.2. By induction on the syntax of C

- If $C = Q'$, then the result follows from Lemma 4.2
- If $C = C_1 \otimes C_2$, then we can prove the result like we proved the corresponding case in Lemma 4.2, using Lemma 5.1.
- If $C = C_1 \,\&\, C_2$, then we the case where $\pi = 1$ is immediate, so we can assume without loss of generality that $\pi = \omega$, and, therefore, that $\pi\cdot C = \pi\cdot C_1 \otimes \pi\cdot C_2$. By Lemma 5.1, we have that $Q \vdash C_1$ and $Q \vdash C_2$; hence, by induction, $\omega\cdot Q \vdash \omega\cdot C_1$ and $\omega\cdot Q \vdash \omega\cdot C_1$. Then, by definition of the entailment relation, we have $\omega\cdot Q \otimes \omega\cdot Q \vdash \omega\cdot C_1 \otimes \omega\cdot C_2$, which concludes, since $\omega\cdot Q = \omega\cdot Q \otimes \omega\cdot Q$.
- If $C = \rho\cdot(Q_1 \multimap C')$, then by Lemma 5.1, there is a Q' such that $Q = \pi\cdot Q'$ and $Q' \otimes Q_1 \vdash C'$. Applying rule C-Impl with $\pi\cdot\rho$, we get $(\pi\cdot\rho)\cdot Q' \vdash (\pi\cdot\rho)\cdot(Q_1 \multimap C')$.
  In other words: $\pi\cdot Q \vdash \pi\cdot(\rho\cdot(Q \multimap C))$ as expected.

□

Proof of Lemma 5.3. By induction on the syntax of C

$$
\begin{cases}
\mathbf{Ds}\,(z; Q; \Gamma \vdash x : v\ [\overline{\tau}\ /\ \overline{a}]) = \\
\quad x\ z \\
\mathbf{Ds}\,(z; Q; \Gamma \vdash \lambda x.e : \tau_1 \to_\pi \tau_2) = \\
\quad \lambda x.\mathbf{Ds}\,(z; Q; \Gamma, x :_\pi \tau_1 \vdash e : \tau_2) \\
\mathbf{Ds}\,(z; Q_1 \otimes Q_2; \Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : \tau) = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, z_2) \to \\
\qquad \mathbf{Ds}\,(z_1; Q_1; \Gamma_1 \vdash e_1 : \tau_1 \to_1 \tau)\ \mathbf{Ds}\,(z_2; Q_2; \Gamma_2 \vdash e_2 : \tau_1)\} \\
\mathbf{Ds}\,(z; Q_1 \otimes \omega \cdot Q_2; \Gamma_1 + \omega \cdot \Gamma_2 \vdash e_1\ e_2 : \tau) = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, \underline{z_2}) \to \\
\qquad \mathbf{Ds}\,(z_1; Q_1; \Gamma_1 \vdash e_1 : \tau_1 \to_\omega \tau)\ \mathbf{Ds}\,(z_2; Q_2; \Gamma_2 \vdash e_2 : \tau_1)\} \\
\mathbf{Ds}\,(z; Q \otimes Q_1[\overline{v}/\overline{a}]; \Gamma \vdash \mathbf{pack}\ e : \exists\ \overline{a}.\tau \multimap Q_1) = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z', z'') \to \\
\qquad \mathbf{pack}\ (z'', \mathbf{Ds}\,(z'; Q; \Gamma \vdash e : \tau[\overline{v}/\overline{a}]))\} \\
\mathbf{Ds}\,(z; Q_1 \otimes Q_2; \Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\ x = e_1\ \mathbf{in}\ e_2 : \tau) = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, z_2) \to \\
\qquad \mathbf{unpack}\ (z', x) = \mathbf{Ds}\,(z'; Q_1; \Gamma_1 \vdash e_1 : \exists\ \overline{a}.\tau_1 \multimap Q)\ \mathbf{in} \\
\qquad \mathbf{let}_1\ z_2{}' = (z_2, z')\ \mathbf{in} \\
\qquad \mathbf{Ds}\,(z_2'; Q_2 \otimes Q; \Gamma_2, x :_1 \tau_1 \vdash e_2 : \tau)\} \\
\mathbf{Ds}\,(z; Q_1 \otimes Q_2; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}_1\ x = e_1\ \mathbf{in}\ e_2 : \tau) = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, z_2) \to \\
\qquad \mathbf{let}_1\ x : \mathbf{Ev}\,(Q) \to_1 \tau_1 = \mathbf{Ds}\,(z_1; Q_1 \otimes Q; \Gamma_1 \vdash e_1 : \tau_1) \\
\qquad \mathbf{in}\ \mathbf{Ds}\,(z_2; Q_2; \Gamma_2, x :_1 \tau_1 \vdash e_2 : \tau)\} \\
\mathbf{Ds}\,(z; \omega \cdot Q_1 \otimes Q_2; \omega \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}_\omega\ x = e_1\ \mathbf{in}\ e_2 : \tau) = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(\underline{z_1}, z_2) \to \\
\qquad \mathbf{let}_\omega\ x : \mathbf{Ev}\,(Q) \to_1 \tau_1 = \mathbf{Ds}\,(z_1; Q_1 \otimes Q; \Gamma_1 \vdash e_1 : \tau_1)\ \mathbf{in} \\
\qquad \mathbf{Ds}\,(z_2; Q_2; \Gamma_2, x :_\omega \tau_1 \vdash e_2 : \tau)\} \\
\mathbf{Ds}\,(z; Q_1 \otimes Q_2; \Gamma_1 + \Gamma_2 \vdash \mathbf{let}_1\ x : \forall\ \overline{a}.Q \multimap \tau_1 = e_1\ \mathbf{in}\ e_2 : \tau) = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, z_2) \to \\
\qquad \mathbf{let}_1\ x : \forall\ \overline{a}.\mathbf{Ev}\,(Q) \to_1 \tau_1 = \mathbf{Ds}\,(z_1; Q_1 \otimes Q; \Gamma_1 \vdash e_1 : \tau_1)\ \mathbf{in} \\
\qquad \mathbf{Ds}\,(z_2; Q_2; \Gamma_2, x :_1 \forall\ \overline{a}.Q \multimap \tau_1 \vdash e_2 : \tau)\} \\
\mathbf{Ds}\,(z; \omega \cdot Q_1 \otimes Q_2; \omega \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}_\omega\ x : \forall\ \overline{a}.Q \multimap \tau_1 = e_1\ \mathbf{in}\ e_2 : \tau) = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(\underline{z_1}, z_2) \to \\
\qquad \mathbf{let}_\omega\ x : \forall\ \overline{a}.\mathbf{Ev}\,(Q) \to_1 \tau_1 = \mathbf{Ds}\,(z_1; Q_1 \otimes Q; \Gamma_1 \vdash e_1 : \tau_1)\ \mathbf{in} \\
\qquad \mathbf{Ds}\,(z_2; Q_2; \Gamma_2, x :_\omega \tau_1 \vdash e_2 : \tau)\} \\
\mathbf{Ds}\,(z; \omega \cdot Q_1 \otimes Q_2; \omega \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_1\ e\ \mathbf{of}\ \{\overline{K_i\ \overline{x_i} \to e_i}\} : \tau) = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(\underline{z_1}, z_2) \to \\
\qquad \mathbf{case}_1\ \mathbf{Ds}\,(z_1; Q_1; \Gamma_1 \vdash e : T\ \overline{\tau})\ \mathbf{of} \\
\qquad\quad \{\overline{K\ \overline{x}_i \to \mathbf{Ds}\,(z_2; Q_2; \Gamma_2, \overline{x_i :_{(\pi \cdot \pi_i)} v_i[\overline{\tau}/\overline{a}]} \vdash e_i : \tau)}\}\} \\
\mathbf{Ds}\,(z; Q_1 \otimes Q_2; \Gamma_1 + \Gamma_2 \vdash \mathbf{case}_\omega\ e\ \mathbf{of}\ \{\overline{K_i\ \overline{x_i} \to e_i}\} : \tau) = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, z_2) \to \\
\qquad \mathbf{case}_\omega\ \mathbf{Ds}\,(z_1; Q_1; \Gamma_1 \vdash e : T\ \overline{\tau})\ \mathbf{of} \\
\qquad\quad \{\overline{K\ \overline{x}_i \to \mathbf{Ds}\,(z_2; Q_2; \Gamma_2, \overline{x_i :_{(\pi \cdot \pi_i)} v_i[\overline{\tau}/\overline{a}]} \vdash e_i : \tau)}\}\}
\end{cases}
$$

Fig. 13. Desugaring

- If $C = Q'$, then the result follows from Lemma 4.3
- If $C = C_1 \otimes C_2$, then we can prove the result like we proved the corresponding case in Lemma 4.3 using Lemma 5.1.
- If $C = C_1 \,\&\, C_2$, then we the case where $\pi = 1$ is immediate, so we can assume without loss of generality that $\pi = \omega$, and, therefore, that $\pi \cdot C = \pi \cdot C_1 \otimes \pi \cdot C_2$. By Lemma 5.1, there exist Q_1 and Q_2 such that $Q_1 \vdash \omega \cdot C_1$, $Q_2 \vdash \omega \cdot C_2$ and $Q = Q_1 \otimes Q_2$. By induction hypothesis, we get $Q_1 = \omega \cdot Q_1'$ and $Q_2 = \omega \cdot Q_2'$ such that $Q_1' \vdash C_1$ and $Q_2' \vdash C_2$. From which it follows that $\omega \cdot Q_1' \otimes \omega \cdot Q_2' \vdash C_1$ and $\omega \cdot Q_1' \otimes \omega \cdot Q_2' \vdash C_1$ (by Lemma B.2) and, finally, $Q = \omega \cdot Q$ (by Lemma B.3) and $Q \vdash C_1 \,\&\, C_2$.
- If $C = \rho \cdot (Q_1 \multimap C')$, then $\pi \cdot C = (\pi \cdot \rho) \cdot (Q_1 \multimap C')$. The result follows immediately by Lemma 5.1.

□

Full proof of Lemma 5.4. By induction on $\Gamma \vdash_{\mathsf{i}} e : \tau \rightsquigarrow C$

E-Var We have
- $x :_1 \forall \overline{a}. Q \multimap v \in \Gamma$
- $\Gamma \vdash_{\mathsf{i}} x : v[\overline{\tau}/\overline{a}] \rightsquigarrow Q[\overline{\tau}/\overline{a}]$
- $Q_g \vdash Q[\overline{\tau}/\overline{a}]$

Therefore, by rules E-Var and E-Sub, it follows immediately that $Q_g; \Gamma \vdash x : v[\overline{\tau}/\overline{a}]$

E-Abs We have
- $\Gamma \vdash_{\mathsf{i}} \lambda x.e : \tau_0 \to_\pi \tau \rightsquigarrow C$
- $Q_g \vdash C$
- $\Gamma, x :_\pi \tau_0 \vdash_{\mathsf{i}} e : \tau \rightsquigarrow C$

By induction hypothesis we have
- $Q_g; \Gamma, x :_\pi \tau_0 \vdash e : \tau$

From which follows that $Q_g; \Gamma \vdash \lambda x.e : \tau_0 \to_\pi \tau$.

E-Let We have
- $\pi \cdot \Gamma_1 + \Gamma_2 \vdash_{\mathsf{i}} \mathbf{let}_\pi\ x = e_1\ \mathbf{in}\ e_2 : \tau \rightsquigarrow \pi \cdot C_1 \otimes C_2$
- $Q_g \vdash \pi \cdot C_1 \otimes C_2$
- $\Gamma_2, x :_\pi \tau_1 \vdash_{\mathsf{i}} e_2 : \tau \rightsquigarrow C_2$
- $\Gamma_1 \vdash_{\mathsf{i}} e_1 : \tau_1 \rightsquigarrow C_1$

By Lemma 5.1, there exist Q_1 and Q_2 such that
- $Q_1 \vdash C_1$
- $Q_2 \vdash C_2$
- $Q_g = \pi \cdot Q_1 \otimes Q_2$

By induction hypothesis we have
- $Q_1; \Gamma_1 \vdash e_1 : \tau_1$
- $Q_2; \Gamma_2, x :_\pi \tau_1 \vdash e_1 : \tau_1$

From which follows that $Q_g; \pi \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}_\pi\ x = e_1\ \mathbf{in}\ e_2 : \tau$.

E-LetSig We have
- $\pi \cdot \Gamma_1 + \Gamma_2 \vdash_{\mathsf{i}} \mathbf{let}_\pi\ x : \forall \overline{a}. Q \multimap \tau_1 = e_1\ \mathbf{in}\ e_2 : \tau \rightsquigarrow C_2 \otimes \pi \cdot (Q \multimap C_1)$
- $Q_g \vdash C_2 \otimes \pi \cdot (Q \multimap C_1)$
- $\Gamma_1 \vdash_{\mathsf{i}} e_1 : \tau_1 \rightsquigarrow C_1$
- $\Gamma_2, x :_\pi \forall \overline{a}. Q \multimap \tau_1 \vdash_{\mathsf{i}} e_2 : \tau \rightsquigarrow C_2$

By Lemma 5.1, there exist Q_1, Q_2 such that
- $Q_2 \vdash C_2$
- $Q_1 \otimes Q \vdash C$
- $Q_g = \pi \cdot Q_1 \otimes Q_2$

By induction hypothesis

- $Q_1 \otimes Q; \Gamma_1 \vdash e_1 : \tau_1$
- $Q_2; \Gamma_2, x :_\pi \forall \overline{a}.Q \Rightarrow \tau_1 \vdash e_2 : \tau$

Hence $Q_g; \pi \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}_\pi\ x : \forall \overline{a}.Q \Rightarrow \tau_1 = e_1\ \mathbf{in}\ e_2 : \tau$

E-App We have

- $\Gamma_1 + \pi \cdot \Gamma_2 \vdash_{\mathsf{i}} e_1\ e_2 : \tau \rightsquigarrow C_1 \otimes \pi \cdot C_2$
- $Q_g \vdash C_1 \otimes \pi \cdot C_2$
- $\Gamma_1 \vdash_{\mathsf{i}} e_1 : \tau_2 \to_\pi \tau \rightsquigarrow C_1$
- $\Gamma_2 \vdash_{\mathsf{i}} e_2 : \tau_2 \rightsquigarrow C_2$

By Lemma 5.1, there exist Q_1, Q_2 such that

- $Q_1 \vdash C_1$
- $Q_2 \vdash C_2$
- $Q_g = Q_1 \otimes \pi \cdot Q_2$

By induction hypothesis

- $Q_1; \Gamma_1 \vdash e_1 : \tau_2 \to_\pi \tau$
- $Q_2; \Gamma_2 \vdash e_2 : \tau_2$

Hence $Q_g; \Gamma_1 + \pi \cdot \Gamma_2 \vdash e_1\ e_2 : \tau$.

E-Pack We have

- $\omega \cdot \Gamma \vdash_{\mathsf{i}} \mathbf{pack}\ e : \exists \overline{a}.\tau \multimap Q \rightsquigarrow \omega \cdot C \otimes Q[\overline{\tau}/\overline{a}]$
- $Q_g \vdash C \otimes Q[\overline{v}/\overline{a}]$
- $\Gamma \vdash_{\mathsf{i}} e : \tau[\overline{v}/\overline{a}] \rightsquigarrow C$

By Lemma 5.1, there exist Q_1, Q_2 such that

- $Q_1 \vdash C$
- $Q_2 \vdash Q[\overline{v}/\overline{a}]$
- $Q_g = \omega \cdot Q_1 \otimes Q_2$

Bu induction hypothesis

- $Q_1; \Gamma \vdash e : \tau[\overline{v}/\overline{a}]$

So we have $Q_1 \otimes Q[\overline{v}/\overline{a}]; \omega \cdot \Gamma \vdash \mathbf{pack}\ e : \exists \overline{a}.\tau \multimap Q$. By rule E-Sub, we conclude $Q_g; \omega \cdot \Gamma \vdash \mathbf{pack}\ e : \exists \overline{a}.\tau \multimap Q$.

E-Unpack We have

- $\Gamma_1 + \Gamma_2 \vdash_{\mathsf{i}} \mathbf{unpack}\ x = e_1\ \mathbf{in}\ e_2 : \tau \rightsquigarrow C_1 \otimes Q' \Rightarrow C_2$
- $Q_g \vdash C_1 \otimes Q' \Rightarrow C_2$
- $\Gamma_1 \vdash_{\mathsf{i}} e_1 : \exists \overline{a}.\tau_1 \multimap Q' \rightsquigarrow C_1$
- $\Gamma_2, x :_\pi \tau_1 \vdash_{\mathsf{i}} e_2 : \tau \rightsquigarrow C_2$

By Lemma 5.1, there exist Q_1, Q_2 such that

- $Q_1 \vdash C_1$
- $Q_2 \otimes Q' \vdash C_2$
- $Q_g = Q_1 \otimes Q_2$

By induction hypothesis

- $Q_1; \Gamma_1 \vdash e_1 : \exists \overline{a}.\tau_1 \multimap Q'$
- $Q_2 \otimes Q; \Gamma_2 \vdash e_2 : \tau$

Therefore $Q_g; \Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\ x = e_1\ \mathbf{in}\ e_2 : \tau$.

E-Case We have

- $\pi \cdot \Gamma + \Delta \vdash_{\mathsf{i}} \mathbf{case}_\pi\ e\ \mathbf{of}\ \{\overline{K_i\ \overline{x_i} \to e_i}\} : \tau \rightsquigarrow \pi \cdot C \otimes \binampersand\ C_i$
- $Q_g \vdash \pi \cdot C \otimes \binampersand\ C_i$
- $\Gamma \vdash_{\mathsf{i}} e : T\ \overline{\sigma} \rightsquigarrow C$
- For each i, $\Delta, \overline{x_i :_{(\pi \cdot \pi_i)} v_i[\overline{\sigma}/\overline{a}]} \vdash_{\mathsf{i}} e_i : \tau \rightsquigarrow C_i$

By repeated uses of Lemma 5.1, there exist Q, Q' such that

- $Q \vdash C$
- For each i, $Q' \vdash C_i$
- $Q_g = \pi \cdot Q \otimes Q'$

By induction hypothesis

- $Q; \Gamma \vdash e : T\,\overline{\sigma}$
- For each i, $Q'; \Delta, \overline{x_i :_{(\pi \cdot \pi_i)} v_i[\overline{\sigma}/\overline{a}]} \vdash e_i : \tau$

Therefore $Q_g; \pi \cdot \Gamma + \Delta \vdash \mathbf{case}_\pi\ e\ \mathbf{of}\ \{\overline{K_i\ \overline{x_i} \to e_i}\} : \tau$.

□

Lemma B.2 (Weakening of wanteds). *If $Q \vdash C$, then $\omega \cdot Q' \otimes Q \vdash C$*

Proof. This is proved by a straightforward induction on the derivation of $Q \vdash C$, using the corresponding property on the simple-constraint entailment relation from Definition 4.1, for the C-Dom case. □

Lemma B.3. *The following equality holds: $\pi \cdot (\rho \cdot C) = (\pi \cdot \rho) \cdot C$.*

Proof. This is proved by a straightforward induction on the structure of C, using Lemma B.1 for the case $C = Q$. □