Linear Constraints

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

A linear 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 GHC's existing constraint solver algorithm. Soundness of linear constraints is ensured by the fact that they desugar into Linear Haskell.

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

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], where IO_L refers to the linear IO monad:

$$
\begin{array}{l}
\mathit{firstLine} :: \mathit{FilePath} \rightarrow \mathit{IO}_L\ \mathit{String} \\
\mathit{firstLine}\ \mathit{fp} = \mathbf{do}\ \{\, h \leftarrow \mathit{openFile}\ \mathit{fp} \\
\qquad\qquad\qquad\quad ;\ (h, \mathit{Ur}\ \mathit{xs}) \leftarrow \mathit{readLine}\ h \\
\qquad\qquad\qquad\quad ;\ \mathit{closeFile}\ h \\
\qquad\qquad\qquad\quad ;\ \mathit{return}\ \mathit{xs} \,\}
\end{array}
$$

This function opens a file, reads its first line, then closes it. Linearity ensures that the file handle h is consumed at the end. Forgetting to call *closeFile* h would result in a type error because 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 arbitrarily many times. Compare the above function with what one would write in a non-linear language:

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

2021. XXXX-XXXX/2021/8-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

$$
\begin{aligned}
&\mathit{firstLine} :: \mathit{FilePath} \rightarrow \mathit{IO}\ \mathit{String} \\
&\mathit{firstLine}\ \mathit{fp} = \mathbf{do}\ \{\, h \leftarrow \mathit{openFile}\ \mathit{fp} \\
&\qquad\qquad\qquad\quad ;\ \mathit{xs} \leftarrow \mathit{readLine}\ h \\
&\qquad\qquad\qquad\quad ;\ \mathit{closeFile}\ h \\
&\qquad\qquad\qquad\quad ;\ \mathit{return}\ \mathit{xs} \,\}
\end{aligned}
$$

This version is less safe, because the type system does not track the linearity of the file handle. However, it is simpler because there is less bureaucracy to manage. We see here a clear tension between extra safety and clarity of code—one we wish, as language designers, to avoid. Reading the non-linear version, it is clear where the handle is used, and ultimately, consumed. How can we get the compiler to see that the handle is used safely without explicit threading?

Rust introduces the *borrow checker* for this very purpose. Our approach is, in some ways, simpler: we show in this paper how a natural generalisation of Haskell's type class constraints does the trick. We call our new constraints *linear constraints*. Like class constraints, linear constraints are propagated implicitly by the compiler. Like linear arguments, they can safely be used to track resources such as file handles. Thus, linear constraints are the combination of these two concepts, which have been studied independently elsewhere [Bernardy et al. 2017; Cervesato et al. 2000; Hodas and Miller 1994; Vytiniotis et al. 2011].

With our extension, we can write a new safe version of *firstLine* which does not require explicit threading of the handle:

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

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 (Section 9.3 suggests how we can get rid of the **pack!**, too). Crucially, the resource representing the open file is a linear constraint, and so it 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 5). This system supports examples that have motivated the design of several resource-aware systems, such as ownership à la Rust (Section 4), or capabilities in the style of Mezzo [Pottier and Protzenko 2013] or ATS [Zhu and Xi 2005]; accordingly, our system 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 6).
- A core language (directly adapted from Linear Haskell [Bernardy et al. 2017]) that supports linear functions. Expressions in our qualified type system desugar into this core language, and we prove that the output of our desugaring is well-typed (Section 7).

Our design is intended to work well with other features of Haskell and its implementation within GHC and we have a prototype implementation.

(a) Linear Types	(b) Linear Constraints
$\mathit{openFile} :: \mathit{FilePath} \to IO_L\ \mathit{Handle}$	$\mathit{openFile} :: \mathit{FilePath} \to IO_L\ (\exists\ h.\mathit{Ur}\ (\mathit{Handle}\ h) \otimes \mathit{Open}\ h)$
$\mathit{readLine} :: \mathit{Handle} \multimap IO_L\ (\mathit{Handle}, \mathit{Ur}\ \mathit{String})$	$\mathit{readLine} :: \mathit{Open}\ h \Rightarrow\!\circ\ \mathit{Handle}\ h \to IO_L\ (\mathit{Ur}\ \mathit{String} \otimes \mathit{Open}\ h)$
$\mathit{closeFile} :: \mathit{Handle} \multimap IO_L\ ()$	$\mathit{closeFile} :: \mathit{Open}\ h \Rightarrow\!\circ\ \mathit{Handle}\ h \to IO_L\ ()$

Fig. 1. Interfaces for file manipulation

2 BACKGROUND: LINEAR HASKELL

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

> *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 then consume its result exactly once.
- To consume a pair exactly once, pattern-match on it, and then consume each component exactly once.
- In general, to consume a value of an algebraic datatype exactly once, pattern-match on it, and then consume all its linear components exactly once.

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 *readLine* example in the introduction 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 giving the illusion that h is threaded around.

From the perspective of the programmer, this is unwanted boilerplate. The approach with linear constraints is to let the handle behave non-linearly, and let its state (i.e., that the handle is open) be a linear constraint. Once this open state is consumed, the handle can no longer be read from or be closed without triggering a compile time error.

3 WORKING WITH LINEAR CONSTRAINTS

Consider the Haskell function *show*:

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

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

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

For our *readLine* example, the implicit argument is the state of a handle *h*. This can be managed as a constraint $Open\ h$, which indicates that the file associated with the handle is open. That is, the constraint $Open\ h$ is provable iff the file associated with *h* is open. This constraint is linear: it is consumed (that is, used as an assumption in a function call) exactly once. In order to manage linearity implicitly, this paper introduces a linear constraint arrow ($\Rrightarrow$), much like Linear Haskell introduces a linear function arrow ($\multimap$). Constraints to the left of a linear constraint arrow are *linear constraints*. Using the linear constraint $Open\ h$, we can give the following type to *closeFile*:

$$closeFile :: Open\ h \Rrightarrow Handle\ h \rightarrow IO_L\ ()$$

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 being closed. Ideally, the linear constraint would refer directly to the handle value, and have the type $closeFile :: (h :: Handle) \rightarrow Open\ h \Rrightarrow IO_L\ ()$. While giving a name to a function argument is common in some dependently typed languages such as ATS [Xi 2017] or Idris, our approach, on the other hand, shows how we can still link a runtime value and a compile-time type variable without needing the full power of dependent types.
- 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.

The above 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[2] a type construction $\exists\ a_1 \ldots a_n.t \otimes Q$, where Q is a linear constraint (scoped over the $a_1 \ldots a_n$) that is paired with the type t.[3] The type of *openFile* is thus:

$$openFile :: FilePath \rightarrow IO_L\ (\exists\ h.Ur\ (Handle\ h) \otimes Open\ h)$$

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

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

[2]There is a variety of ways existential types can be worked into a language. The existentials we use here might best be understood as a mild generalisation of those presented by Pierce [2002, Chapter 24]. However a recent publication by Eisenberg et al. [2021] works out an approach which could make linear constraint easier to use, as we discuss in Section 9.3.

[3]We will freely omit the $\exists\ a_1 \ldots a_n.$ or $\otimes Q$ parts when they are empty.

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*.

$$readLine :: Open\ h \multimap Handle\ h \to IO_L\ (Ur\ String \otimes Open\ h)$$

Haskell does not have an existential quantifier. However, existentially quantified types can be encoded as a GADTs. For instance, $\exists\ h.Ur\ (Handle\ h) \otimes Open\ h$ can be implemented as

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

In our implementation (Section 8.1), packed linear constraints piggy-back on GHC's standard GADT syntax. Correspondingly, existential types are introduced by a data constructor, which we write as **pack**.

When pattern-matching on a **pack** constructor, all existentially quantified names are brought into scope and all the returned constraints are assumed. We also use **pack**! x as a shorthand for **pack** $(Ur\ x)$. We have now seen all the ingredients needed to write the *firstLine* example as in the introduction.

3.1 Minimal Examples

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

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

$$useC :: C \multimap Int$$

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

3.1.1 Dithering. We reject this program:

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

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

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

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

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

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

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

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

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

3.1.3 Overusing. Finally, consider the following program, which is rejected:

$$\begin{aligned}&overusing :: C \Rrightarrow\!\circ (Int, Int)\\&overusing = (useC, useC)\end{aligned}$$

This program is rejected because it uses C twice.

3.2 Linear I/O

The file-handling example discussed in sections 2 and 3 uses a linear version of the IO monad, IO_L. Compared to the traditional IO monad, the type of the monadic operations $\ggg$ (aka *bind*) and *return* are changed to use linear arrows.

$$\begin{aligned}&(\ggg) :: IO_L\ a \multimap (a \multimap IO_L\ b) \multimap IO_L\ b\\&return :: a \multimap IO_L\ a\end{aligned}$$

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

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

If bind were not linear, the first occurrence of $readLine\ h$ would not be able to consume the $Open\ h$ constraint linearly.

4 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 is not always desirable, either due to its (unpredictable) runtime costs, or because pointers exist between separately-managed memory spaces (for example when calling foreign functions [Domínguez 2020]). In either case, one must then resort to explicit memory allocation and deallocation. This task is error prone: one can easily forget a deallocation (causing a memory leak) or deallocate several times (corrupting data).

4.1 Ownership constraints

With linear constraints, it is possible to provide a *pure* interface for manual memory management, which enforces correct deallocation of memory. Our approach, in the style of Rust, is to use a linear constraint to represent *ownership* of a memory location. 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 $Own\ n$, n is a type variable (of a special kind *Location*) which represents a memory location. Locations mediate the relationship between references and ownership constraints. In fact, for more granular control, we will be using three linear constraints: one for reading, one for writing, and one for ownership.

$$\mathbf{class}\ Read\ (n :: Location) \qquad \mathbf{class}\ Write\ (n :: Location) \qquad \mathbf{class}\ Own\ (n :: Location)$$

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 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 three 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 defeat the point of *O n* be a linear 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.

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

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

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

Instead of deallocating the reference, one could transfer control of the memory location to the garbage collector. This operation is sometimes called "freezing":

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

4.2 Arrays

The above toolkit handles references to base types just fine. But what about storing references in objects managed by the ownership system? In this section, we show how to deal with arrays of references, (including arrays of arrays), as a typical case.

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

For this purpose we introduce the type $\mathit{PArray}\ a\ n$, where the kind of a is $\mathit{Location} \rightarrow \mathit{Type}$: this way we can easily enforce that each reference in the array refers to the same location n. Both types $\mathit{AtomRef}\ \ a$ and $\mathit{PArray}\ \ a$ have kind $\mathit{Location} \rightarrow \mathit{Type}$, and therefore one can allocate, and manipulate arrays of arrays with this API.

When writing a reference (be it an array or an $\mathit{AtomRef}$) in an array, ownership of the reference is transferred to the array.

$$\mathit{writePArray} :: (\mathit{RW}\ \ n, O\ p) \Rrightarrow \mathit{PArray}\ a\ n \rightarrow \mathit{Int} \rightarrow a\ p \rightarrow () \otimes \mathit{RW}\ \ n$$

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 references in the array so we provide $\mathit{lendPArrayElt}$:

$$\mathit{lendPArrayElt} :: \mathit{RW}\ \ n \Rrightarrow \mathit{PArray}\ a\ n \rightarrow \mathit{Int} \rightarrow (\forall\ p.\ \mathit{RW}\ \ p \Rrightarrow a\ p \rightarrow r \otimes \mathit{RW}\ \ p) \multimap r \otimes \mathit{RW}\ \ n$$

The $\mathit{lendPArrayElt}\ \ a\ \ i\ \ k$ 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 $\mathit{RW}\ \ p$ 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, $\mathit{RW}\ \ n$ and $\mathit{RW}\ \ p$ are never simultaneously available. The following function, on the other hand, would not be sound:

$$\mathit{extractEltWrong} :: \mathit{RW}\ \ n \Rrightarrow \mathit{PArray}\ a\ n \rightarrow \mathit{Int} \rightarrow \exists\ p. \mathit{Ur}\ (a\ p) \otimes (\mathit{RW}\ \ n, \mathit{RW}\ \ p)$$

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

$$\begin{aligned} \mathit{lendPArrayEltRead}\ &::\ \mathit{Read}\ \ n \Rrightarrow \mathit{PArray}\ a\ n \rightarrow \mathit{Int} \\ &\rightarrow (\forall\ p.\ \mathit{Read}\ \ p \Rrightarrow a\ p \rightarrow r \otimes \mathit{Read}\ \ p) \\ &\multimap r \otimes \mathit{Read}\ \ n \end{aligned}$$

Finally, we can freeze arrays, using the following primitive:

$$\mathit{freezePArray} :: O\ \ n \Rrightarrow \mathit{PArray}\ a\ n \rightarrow () \otimes \omega \cdot (\mathit{Read}\ \ n)$$

Where $\omega \cdot (\mathit{Read}\ \ n)$ means that the returned constraint is not linear. That is, after $\mathit{freezePArray}\ \ n$, we have unrestricted read access to n (and any element of n), as expected. We describe this syntax in more details in the next section, where, similarly, we shall treat $\mathit{Read}\ \ n \Rightarrow$ as an abbreviation for $\omega \cdot (\mathit{Read}\ \ n) \Rrightarrow$.

With an unrestricted $\mathit{Read}\ \ n$ capability, we can read from the array more directly with the following primitive

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

5 A QUALIFIED TYPE SYSTEM FOR LINEAR CONSTRAINTS

Having laid out situations which would benefit from linear constraints, we now present our design for a qualified type system [Jones 1994] that supports them. Our design is mindful of our goal of integration with Haskell and GHC and is thus based on the work of Vytiniotis et al. [2011], which undergirds GHC's current approach to type inference.

5.1 Multiplicities

Like in Linear Haskell we make use of a system of *multiplicities*, which describe how many times a function consumes its input. As multiplicities are central to our constraint calculus, we will colour them in blue like constraints. 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).

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

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 the semiring operations: addition and multiplication. Here and in the rest of the paper we adopt the convention that equations defining a function by pattern matching are marked with a { to their left.

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

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

5.2 Simple Constraints and Entailment

Let us now turn to constraints themselves. We call constraints such as $Read\ n$ or $Write\ n$ *atomic constraints*. The exact nature of atomic constraints is left unspecified: the set of atomic constraints is a parameter of our qualified type system.

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

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

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

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

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

$Q \Vdash Q$
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 $Q \Vdash \rho \cdot q$ then $\pi \cdot Q \Vdash (\pi \cdot \rho) \cdot q$
if $Q \Vdash (\pi \cdot \rho) \cdot q$ then there exists Q' such that $Q = \pi \cdot Q'$ and $Q' \Vdash \rho \cdot q$
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 $Q_1 \Vdash Q_2$

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

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

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

Definition 5.2 (Simple constraints).

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

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

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

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

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

Definition 5.3 (Entailment relation). The qualified type system is parameterised by a relation $Q_1 \Vdash Q_2$ between two simple constraints. The entailment relation must obey the laws listed in Figure 2.

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

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

Then the following properties hold

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

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

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

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

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

Fig. 3. Grammar of the qualified type system

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

Proofs of these lemmas (and others) appear in our anonymised supplementary material; they can be proved by straightforward use of the properties in Figure 2.

5.3 Typing rules

With this material in place, we can now present our type system. The grammar is given in Figure 3, which also includes the definitions of scaling on contexts $\pi \cdot \Gamma$ and addition of contexts $\Gamma_1 + \Gamma_2$. Note that addition on contexts is actually a partial function, as it requires that, if a variable x is bound in both Γ_1 and Γ_2, then x is assigned the same type in both (but perhaps different multiplicities). This partiality is not a problem in practice, as the required condition for combining contexts is always satisfied.

The typing rules are in Figure 4. A qualified type system [Jones 1994] such as ours introduces a judgement of the form $Q; \Gamma \vdash e : \tau$, where Γ is a standard type context, and Q is a constraint we have assumed to be true. Q behaves much like Γ, which will be instrumental for desugaring in Section 7; the main difference is that Γ is addressed explicitly, whereas Q is used implicitly in rule E-Var.

The type system of Figure 4 is purely declarative: note, for example, that rule E-App does not describe how to break the typing assumptions into constraints Q_1/Q_2 and contexts Γ_1/Γ_2. We will see how to infer constraints in Section 6. Yet, this system is our ground truth: a system with a simple enough definition that programmers can reason about typing. We do not directly give a dynamic semantics to this language; instead, we will give it meaning via desugaring to a simpler core language in Section 7.

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

Linear functions. The type of linear functions is written $a \to_1 b$. Despite our focus on linear constraints, we still need linearity in ordinary arguments. For example, the bind combinator for IO_L (Section 1) requires linear arrows: $(\mathbin{>\!\!>\!=}) :: IO_L\ a \multimap (a \multimap IO_L\ b) \multimap IO_L\ b$.

Indeed, the linearity of arrows interacts in interesting ways with linear constraints: If $f : a \to_\omega b$ and $x : 1 \cdot q \mathrel{=\!\!\circ} a$, then calling $f\ x$ would actually use q many times. We must make sure it is impossible to derive $1 \cdot q; f{:}_\omega a \to_\omega b, x{:}_\omega 1 \cdot q \mathrel{=\!\!\circ} a \vdash f\ x : b$. Otherwise we could make, for instance,

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

$$\textsc{E-Var}\quad \frac{\Gamma_1 = x{:}_1 \forall\overline{a}.Q_1 \mathrel{=\!\!\circ} v}{Q_1[\overline{\tau}/\overline{a}];\Gamma_1 + \omega\cdot\Gamma_2 \vdash x : v[\overline{\tau}/\overline{a}]}
\qquad
\textsc{E-Abs}\quad \frac{Q;\Gamma, x{:}_\pi \tau_1 \vdash e : \tau_2}{Q;\Gamma \vdash \lambda x.e : \tau_1 \to_\pi \tau_2}
\qquad
\textsc{E-App}\quad \frac{Q_1;\Gamma_1 \vdash e_1 : \tau_1 \to_\pi \tau \qquad Q_2;\Gamma_2 \vdash e_2 : \tau_1}{Q_1 \otimes \pi\cdot Q_2;\Gamma_1 + \pi\cdot\Gamma_2 \vdash e_1\ e_2 : \tau}$$

$$\textsc{E-Pack}\quad \frac{Q;\Gamma \vdash e : \tau[\overline{v}/\overline{a}]}{Q \otimes Q_1[\overline{v}/\overline{a}];\Gamma \vdash \mathbf{pack}\ e : \exists\overline{a}.\tau \obslash Q_1}
\qquad
\textsc{E-Unpack}\quad \frac{Q_1;\Gamma_1 \vdash e_1 : \exists\overline{a}.\tau_1 \obslash Q \qquad \overline{a}\ \text{fresh} \qquad Q_2 \otimes Q;\Gamma_2, x{:}_1\tau_1 \vdash e_2 : \tau}{Q_1 \otimes Q_2;\Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\ x = e_1\ \mathbf{in}\ e_2 : \tau}$$

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

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

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

Fig. 4. Qualified type system

the *overusing* function from Section 3.1.3. You can check that $1\cdot q; f{:}_\omega a \to_\omega b, x{:}_\omega 1\cdot q \mathrel{=\!\!\circ} a \vdash f\ x : b$ indeed does not type check, because the scaling of Q_2 in rule E-App ensures that the constraint would be $\omega\cdot q$ instead. On the other hand, it is perfectly fine to have $1\cdot q; f{:}_\omega a \to_1 b, x{:}_\omega 1\cdot q \mathrel{=\!\!\circ} a \vdash f\ x : b$ when f is a linear function.

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

Local assumptions. Rule E-Let includes support for local assumptions. We thus have the ability to generalise a subset of the constraints needed by e_1 (but not the type variables—no **let**-generalisation here, though it could be added). The inference algorithm of Section 6 will not make use of this possibility, but we revisit this capability in Section 9.1.

Existentials. We include $\exists\overline{a}.\tau \obslash Q$, as introduced in Section 3, together with the **pack** and **unpack** constructions. See rules E-Pack and E-Unpack.

6 CONSTRAINT INFERENCE

The type system of Figure 4 gives a declarative description of what programs are acceptable. We now present the algorithmic counterpart to this system. Our algorithm is structured, unsurprisingly, around generating and solving constraints, broadly following the template of Pottier and Rémy [2005]. That is, our algorithm takes a pass over the abstract syntax entered by the user, generating constraints as it goes. Then, separately, we solve those constraints (that is, try to satisfy them) in the presence of a set of assumptions, or we determine that the assumptions do not imply that the constraints hold. In the latter case, we issue an error to the programmer.

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

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

6.1 Wanted constraints

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

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

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

The last form of wanted constraint C is an implication $\pi \cdot (Q \multimap C)$. Proving $\pi \cdot (Q \multimap C)$ allows us to assume Q linearly while proving C, a total of π times. These implications arise when we unpack an existential package that contains a linear constraint and also when checking a **let**-binding. We can define scaling over wanted constraints by recursion as follows, where we use scaling over

$\boxed{Q \vdash C}$ *(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

simple constraints in the simple-constraint case:

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

For the most part, scaling of wanted constraints is straightforward. The only peculiar case is when we scale the additive conjunction $C_1 \,\&\, C_2$ by ω, the result is a multiplicative conjunction. The intuition here is that when proving $\omega{\cdot}(C_1 \,\&\, C_2)$, we need to have unrestricted access to the assumptions of both C_1 and C_2.

We define an entailment relation over wanteds in Figure 5. Note that this relation uses only simple constraints Q as assumptions, as there is no way to assume the more elaborate C[4].

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

Lemma 6.1 (Inversion). *The inference rules of $Q \vdash C$ can be read bottom-up as well as top-down, as is required of $Q_1 \Vdash Q_2$ in Figure 2. 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 $Q = \pi{\cdot}Q_1$*

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

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

6.2 Constraint generation

The process of inferring constraints is split into two parts: generating constraints, which we do in this section, then solving them in Section 6.3. Constraint generation is described by the judgement $\Gamma \mapsto e : \tau \rightsquigarrow C$ (defined in Figure 6) which outputs a constraint C required to make e typecheck. The definition $\Gamma \mapsto e : \tau \rightsquigarrow C$ is syntax directed, so it can directly be read as an algorithm, taking as input a *typing derivation* for $\Gamma \vdash e : \tau$ (produced by an external type inference oracle as discussed above). Notably, the algorithm has access to the context splitting from the (previously computed) typing derivation, and is thus indeed syntax directed.

The rules of Figure 6 constitute a mostly unsurprising translation of the rules of Figure 4, except for the following points of interest:

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

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

$\boxed{\Gamma \mathrel{|\!\!\blacktriangleright} e : \tau \rightsquigarrow C}$ *(Constraint generation)*

$$\textsc{G-Var}\quad \frac{\Gamma_1 = x{:_1}\forall\overline{a}.Q \mathrel{=\!\!\circ} \upsilon}{\Gamma_1 + \omega\cdot\Gamma_2 \mathrel{|\!\!\blacktriangleright} x : \upsilon[\overline{\tau}/\overline{a}] \rightsquigarrow Q[\overline{\tau}/\overline{a}]}$$

$$\textsc{G-Abs}\quad \frac{\Gamma, x{:_\pi}\tau_0 \mathrel{|\!\!\blacktriangleright} e : \tau \rightsquigarrow C}{\Gamma \mathrel{|\!\!\blacktriangleright} \lambda x.e : \tau_0 \to_\pi \tau \rightsquigarrow C}$$

$$\textsc{G-App}\quad \frac{\Gamma_1 \mathrel{|\!\!\blacktriangleright} e_1 : \tau_2 \to_\pi \tau \rightsquigarrow C_1 \qquad \Gamma_2 \mathrel{|\!\!\blacktriangleright} e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma_1 + \pi\cdot\Gamma_2 \mathrel{|\!\!\blacktriangleright} e_1\, e_2 : \tau \rightsquigarrow C_1 \otimes \pi\cdot C_2}$$

$$\textsc{G-Pack}\quad \frac{\Gamma \mathrel{|\!\!\blacktriangleright} e : \tau[\overline{\upsilon}/\overline{a}] \rightsquigarrow C}{\Gamma \mathrel{|\!\!\blacktriangleright} \mathbf{pack}\, e : \exists\overline{a}.\tau \oslash Q \rightsquigarrow C \otimes Q[\overline{\upsilon}/\overline{a}]}$$

$$\textsc{G-Unpack}\quad \frac{\Gamma_1 \mathrel{|\!\!\blacktriangleright} e_1 : \exists\overline{a}.\tau_1 \oslash Q_1 \rightsquigarrow C_1 \qquad \overline{a}\ \text{fresh} \qquad \Gamma_2, x{:_1}\tau_1 \mathrel{|\!\!\blacktriangleright} e_2 : \tau \rightsquigarrow C_2}{\Gamma_1 + \Gamma_2 \mathrel{|\!\!\blacktriangleright} \mathbf{unpack}\, x = e_1\, \mathbf{in}\, e_2 : \tau \rightsquigarrow C_1 \otimes 1\cdot(Q_1 \mathrel{=\!\!\circ} C_2)}$$

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

$$\textsc{G-Let}\quad \frac{\Gamma_1 \mathrel{|\!\!\blacktriangleright} e_1 : \tau_1 \rightsquigarrow C_1 \qquad \Gamma_2, x{:_\pi}\tau_1 \mathrel{|\!\!\blacktriangleright} e_2 : \tau \rightsquigarrow C_2}{\pi\cdot\Gamma_1 + \Gamma_2 \mathrel{|\!\!\blacktriangleright} \mathbf{let}_\pi\, x = e_1\, \mathbf{in}\, e_2 : \tau \rightsquigarrow \pi\cdot C_1 \otimes C_2}$$

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

Fig. 6. Constraint generation

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

Unannotated **let**. However, the G-Let rule does not produce an implication constraint, as we do not model **let**-generalisation [Vytiniotis et al. 2010]. Section 9.1 discusses this design choice further.

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

Lemma 6.4 (Soundness of constraint generation). *For all Q_g, if $\Gamma \mathrel{|\!\!\blacktriangleright} e : \tau \rightsquigarrow C$ and $Q_g \vdash C$ then $Q_g; \Gamma \vdash e : \tau$.*

6.3 Constraint solving

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

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

The judgement takes in two contexts: U, which holds all the unrestricted atomic constraint assumptions and L_i, which holds all the linear atomic constraint assumptions. The linear contexts L_i and L_o have been described as multisets (Section 5.2), but we treat them as ordered lists in the more concrete setting here; we will see soon why this treatment is necessary.

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

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

$$\textsc{S-ImplOne}\ \frac{U \cup U_0; L_i \uplus L_0 \vdash_{\mathsf{s}} C \rightsquigarrow L_o \quad L_o \subseteq L_i}{U; L_i \vdash_{\mathsf{s}} 1{\cdot}((U_0, L_0) \multimap C) \rightsquigarrow L_o}$$

$$\textsc{S-Add}\ \frac{U; L_i \vdash_{\mathsf{s}} C_1 \rightsquigarrow L_o \quad U; L_i \vdash_{\mathsf{s}} C_2 \rightsquigarrow L_o}{U; L_i \vdash_{\mathsf{s}} C_1 \,\&\, C_2 \rightsquigarrow L_o} \qquad \textsc{S-ImplMany}\ \frac{U \cup U_0; L_0 \vdash_{\mathsf{s}} C \rightsquigarrow \emptyset}{U; L_i \vdash_{\mathsf{s}} \omega{\cdot}((U_0, L_0) \multimap C) \rightsquigarrow L_i}$$

Fig. 7. Constraint solver

Linearity requires treating constraints as consumable resources. This is what L_o is for: it contains the hypotheses of L_i which are not consumed when proving C. As suggested by the notation, it is an output of the algorithm.

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

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

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

To handle simple wanted constraints, we will need a domain-specific *atomic-constraint solver* to be the algorithmic counterpart of the abstract entailment relation of Section 5.2. The main solver will appeal to this atomic-constraint solver when solving atomic constraints. The atomic-constraint solver is represented by the following judgement:

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

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

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

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

6.3.1 Constraint solver algorithm. Building on this atomic-constraint solver, we use a linear proof search algorithm based on the recipe given by Cervesato et al. [2000]. Figure 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 constraints 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 implications, respectively. In both cases, the assumption of the implication is split into its unrestricted and linear components. When solving a linear implication, we union the assumptions with their respective context, and proceed with solving the conclusion. Importantly

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

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

(a) Entailment relation

$\boxed{U; L \vdash_{\mathsf{s}}^{\mathsf{atom}} \pi{\cdot}q \leadsto L_o}$ *(Atomic-constraint solver)*

$$\text{Atom-Many}\ \frac{q \in U}{U; L \vdash_{\mathsf{s}}^{\mathsf{atom}} \omega{\cdot}q \leadsto L} \qquad \text{Atom-OneL}\ \frac{q \notin L_2 \quad q \notin U}{U; L_1, q, L_2 \vdash_{\mathsf{s}}^{\mathsf{atom}} 1{\cdot}q \leadsto L_1, L_2} \qquad \text{Atom-OneU}\ \frac{q \in U \quad q \notin L}{U; L \vdash_{\mathsf{s}}^{\mathsf{atom}} 1{\cdot}q \leadsto L}$$

(b) Atomic-constraint solver

Fig. 8. A stripped-down constraint domain

(see Section 6.3.2 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 does not leak it to the ambient context. Solving unrestricted implications allows only 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 use their own assumption linearly, but use everything from the ambient context ω times.

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

For the sake of our examples, we need very little: linear constraints can remain abstract. It is thus sufficient for the entailment relation (Figure 8a) to prove q if and only if it is already assumed—while respecting linearity.

The corresponding atomic-constraint solver (Figure 8b) is more interesting. It is deterministic: in all circumstances, only one of the three rules can apply. This means that the algorithm does not guess, thus never needs to backtrack. Avoiding guesses is a key property of GHC's solver [Vytiniotis et al. 2011, Section 6.4], one we must maintain if we are to be compatible with GHC.

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

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

In this example, the programmer meant for *closeFile* to use the *Open h* constraint introduced locally in the type of *cl*. Yet there are actually two *Open h* constraints: this local one and the one assumed in the unpacking of the result of *openFile*. The wrong choice among the constraints will lead the algorithm to fail. Choosing the first q linear assumption guarantees we get the most local one.

Another interesting feature of the solver (Figure 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:

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

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

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

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

Fig. 9. Core calculus (subset)

$\mathbf{class}\ C$
$giveC :: (C \Rightarrow Int) \rightarrow Int$
$useC :: C \Rightarrow\!\circ\ Int$

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

$bad :: C \Rightarrow\!\circ\ (Int, Int)$
$bad = (giveC\ useC, useC)$

It is possible to give a type derivation to *bad* in the qualified type system of Section 5. In this case, the constraint assignment is actually unambiguous: the first *useC* must use the unrestricted C, while the second must use the linear C. This assignment, however, would require the constraint solver to guess when solving the constraint from the first *useC*. Accordingly, in order to both avoid backtracking and to keep type inference independent of the order terms appear in the program text, *bad* is rejected. This introduces incompleteness with respect the entailment relation. We conjecture that this is the only source of incompleteness that we introduce beyond what is already in GHC [Vytiniotis et al. 2011, Section 6].

7 DESUGARING

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

7.1 The core calculus

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

- $\tau_1 \otimes \tau_2$ with sole constructor $(,) : \forall a\, b.a \rightarrow_1 b \rightarrow_1 a \otimes b$. 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 $\mathbf{Ur} : \forall a.a \rightarrow_\omega \mathbf{Ur}\ a$

Figure 9 highlights the differences from the qualified system:

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

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

- We do not support multiplicity polymorphism.
- On the other hand, we do include type polymorphism.
- Polymorphism is implicit rather than explicit. This is not an essential difference, but it simplifies the presentation. We could, for example, include more details in the terms in order to make type-checking more obvious; this amounts essentially to an encoding of typing derivations in the terms[5].
- We have existential types. These can be realised in regular Haskell as a family of datatypes.

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

7.2 From qualified to core

7.2.1 Evidence. In order to desugar derivations of the qualified system to the core calculus, we pass evidence explicitly[6]. To do so, we require some more material from constraints. Namely, we assume a type $[\![q]\!]^{\mathbf{ev}}$ for each atomic constraint q, defined in Figure 10a. The $[\![\_]\!]^{\mathbf{ev}}$ operation extends to simple constraints as $[\![Q]\!]^{\mathbf{ev}}$. Furthermore, we require that for every Q_1 and Q_2 such that $Q_1 \Vdash Q_2$, there is a (linear) function $[\![Q_1 \Vdash Q_2]\!]^{\mathbf{ev}} : [\![Q_1]\!]^{\mathbf{ev}} \to_1 [\![Q_2]\!]^{\mathbf{ev}}$.

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

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

$$\left\{ [\![\forall \overline{a}. Q \Rightarrow\!\!\circ\, \tau]\!] = \forall \overline{a}. [\![Q]\!]^{\mathbf{ev}} \to_1 [\![\tau]\!] \right. \qquad \left\{ \begin{array}{lcl} [\![\bullet]\!] & = & \bullet \\ [\![\Gamma, x{:}_\pi \tau]\!] & = & [\![\Gamma]\!], x{:}_\pi [\![\tau]\!] \end{array} \right.$$

$$\left\{ \begin{array}{lcl} [\![\tau_1 \to_\pi \tau_2]\!] & = & [\![\tau_1]\!] \to_\pi [\![\tau_2]\!] \\ [\![\exists \overline{a}. \tau \otimes Q]\!] & = & \exists \overline{a}. [\![\tau]\!] \otimes [\![Q]\!]^{\mathbf{ev}} \end{array} \right.$$

7.2.3 Translating terms. Given a derivation $Q; \Gamma \vdash e : \tau$, we can build an expression $[\![Q; \Gamma \vdash e : \tau]\!]_z$, such that $[\![\Gamma]\!], z{:}_1[\![Q]\!]^{\mathbf{ev}} \vdash [\![Q; \Gamma \vdash e : \tau]\!]_z : [\![\tau]\!]$ (for some fresh variable z). Even though we abbreviate the derivation as only its concluding judgement, the translation is defined recursively on the whole typing derivation: in particular, we have access to typing rule premises in the body of the definition. We present some of the interesting cases in Figure 10b.

The cases correspond to the E-Var, E-Unpack[7], 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** requires splitting the context into two: e_1 is desugared as a pair, and the evidence it contains is passed to e_2. Finally, subsumption summons the function corresponding to the entailment relation $Q \Vdash Q_1$ and applies it to $z : [\![Q]\!]^{\mathbf{ev}}$ then proceeds to desugar e with the resulting evidence for Q_1. Crucially, since $[\![\_]\!]_z$ is defined on *derivations*, we can access the premises used in the rule. Namely, Q_1 is available in this last case from the E-Sub rule's premise.

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

[5] See, for example, Weirich et al. [2017] and their comparison between an implicit core language D and an explicit one DC.

[6] This technique is also often called dictionary-passing style [Hall et al. 1996] because, in the case of type classes, evidences are dictionaries, and because type classes were the original form of constraints in Haskell.

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

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

(a) Evidence passing

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

(b) Desugaring (subset)

Fig. 10. Evidence passing and desugaring

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

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

8 INTEGRATING INTO GHC

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

8.1 Implementation

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

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

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

As constraint solving proceeds, the compiler pipeline constructs a term in an typed language known as GHC Core [Sulzmann et al. 2007]. In Core, type class constraints are turned into explicit evidence (see Section 7). Thanks to being fully annotated, Core has decidable typechecking which is useful in debugging modifications to the compiler. Thus, the soundness of our implementation is verified by the Core typechecker, which already supports linearity.

8.2 Interaction with other features

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

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

```
class Eq a ⇒ Ord a where
  ...
```

which means that every ordered type must also support equality. Such superclass declarations extend the entailment relation: if we know that a type is ordered, we also know that it supports equality. This is troublesome if we have a linear occurrence of *Ord a*, because then using this entailment, we could conclude that a linear constraint (*Ord a*) implies an unrestricted constraint (*Eq a*), which violates Lemma 5.5.

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

```
class Eq a =∘ Ord a where
  ...
```

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

To address both of these issues at once, we make the following rule: the superclasses of a linear constraint are *never* expanded.

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

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

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

9 EXTENSIONS

The system presented in Sections 5 and 6 is already capable of supporting the examples in Sections 3 and 4. In this section, we consider some potential avenues for extensions.

9.1 let generalisation

As discussed in Section 6.2, the G-LET rule of our constraint generator does not generalise the type of **let**-bindings, which 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 (*i.e.* 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:

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

This program looks reasonable; however, it is rejected. The type of *closeOp* is $IO_L\ ()$, which means that the definition of *closeOp* consumes the linear constraint *Open h*. So, by the time we attempt *readLine h*, the constraint is no longer available.

What the programmer really meant, here, was for *closeOp* to have type $Open\ h \Rrightarrow IO_L\ ()$. After all, a **let** definition is not part of the sequence of instructions: it is just a definition for later, not intended to consume the current state of the file. With no **let**-generalisation, the only way to give *closeOp* the type $Open\ h \Rrightarrow IO_L\ ()$ is to give *closeOp* a type signature. In current GHC, we can't write that signature down, since there is no syntax to bind the type variable h in the program text[8]. But even ignoring this, it would be rather unfortunate if the default behaviour of **let**, in the presence of linear constraints, almost never was what the programmer wants.

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

$$\textsc{G-LetGen}\quad \frac{\Gamma_1 \mapsto e_1 : \tau_1 \rightsquigarrow C_1 \qquad Q_r \otimes Q \vdash C_1 \qquad \Gamma_2, x{:}_{\pi} Q \Rrightarrow \tau_1 \mapsto e_2 : \tau \rightsquigarrow C_2}{\pi{\cdot}\Gamma_1 + \Gamma_2 \mapsto \mathbf{let}_{\pi}\ x = e_1\ \mathbf{in}\ e_2 : \tau \rightsquigarrow \pi{\cdot}Q_r \otimes C_2}$$

This rule is non-deterministic, because it requires finding Q_r and Q. We can modify the constraint solver of Section 6.3 to find $Q_r \otimes Q$, but we still have to split the residual into Q_r and Q somehow.

Any predictable strategy would do: as long as our rule 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.

9.2 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.

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

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

That is $\top$ is unconditionally true, and can consume any number of linear given constraints— indeed, the corresponding program is already crashing. The C-Top rule thus induces a considerable amount of non-determinism in the constraint solver. 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 6.3 without any technical difficulty. We chose, however, to keep empty cases out the presentation because they have 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 $\top$.

[8] Eisenberg et al. [2018] describe a way to fix this shortcoming.

9.3 Inferring pack and unpack

Recent work [Eisenberg et al. 2021] describes an algorithm (call it EDWL, after the authors' names) that can infer the location of **pack**s and **unpack**s[9] in a user's program. In Section 9.2 of that paper, the authors extend their system to include class constraints, much as we allow our existential packages to carry linear constraints.

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

$$
\begin{array}{l}
\mathit{writeTwo} :: \mathit{RW}\ n \multimap \mathit{AtomRef}\ a\ n \to a \to a \to () \otimes \mathit{RW}\ n \\
\mathit{writeTwo}\ \mathit{ref}\ \mathit{val1}\ \mathit{val2} = \mathbf{let}\ \mathit{unit1} = \mathit{writeRef}\ \mathit{ref}\ \mathit{val1}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\ \mathbf{let}\ \mathit{unit2} = \mathit{writeRef}\ \mathit{ref}\ \mathit{val2}\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\ ()
\end{array}
$$

10 RELATED WORK

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

The solver judgement in OutsideIn takes the following form:

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

The main differences from our solver judgement in Section 6.3 are:

- OutsideIn's judgement includes top-level axioms schemes separately ($\mathcal{Q}$), which we have omitted for the sake of brevity and are instead included in Q_{given}.
- We present the *given* constraints (Q_{given} in OutsideIn) as two separate constraint sets U and L, standing for the unrestricted and linear parts respectively.
- In addition to constraint inference, OutsideIn performs type inference, requiring additional bookkeeping in the solver judgment. The solver takes as input a set of *touchable* variables $\overline{\alpha}_{tch}$ which record the type variables that can be unified at any given time, and produces a type substitution θ as an output (whose domain is a subset of the touchable variables). As discussed in Section 6, we do not perform type inference, only constraint inference. Therefore, our solver need not return a type assignment.
- The most important difference is the output of the algorithms. Both OutsideIn and our solver output a set of constraints, $Q_{residual}$ and L_o respectively. However, the meaning of these contexts is different. OutsideIn's *residual* constraints $Q_{residual}$ correspond to the part of C_{wanted} that could not be solved from the assumptions. These residuals are then quantified over in the generalisation step of the inference algorithm. We omit these residuals, which means

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

that our algorithm cannot infer qualified types (though see Section 9.1). Our *output* constraints L_o instead correspond to the part of the *linear* givens L_i that were not used in the solution for C_w. Keeping track of these unused constraints is crucial: ultimately we need to make sure that every linear constraint is used.

- Finally, while OutsideIn has a single kind of conjunction, our constraint language requires two: $Q_1 \otimes Q_2$ and $Q_1 \,\&\, Q_2$. This shows up when generating constraints for **case** expressions in the rule G-Case rule. OutsideIn accumulates constraints across branches (taking the union of each branch), whereas we need to make sure that each branch of a **case**-expression consumes the same constraints. 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 must be closed in the other branches, too. Otherwise, its state will be unknown to the type system after the **case**.

Rust. The memory ownership example of Section 4 is strongly inspired by Rust. 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 this paper, 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 do not have the convenience of Rust's syntax, we expect that they will support a greater variety of abstractions. Even though Rust programmers have come up with varied abstractions which leverage the borrowing mechanism to support applications going beyond memory management (for instance, safe file handling), it is unclear that all applications supported by linear constraints are reducible to the borrowing mechanism.

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. ATS is the closest to our system because it appeals explicitly to linear logic, and because the capabilities (known as *stateful views*) are not tied to a particular use case. However, ATS does not have full inference of capabilities.

Other than that, the two systems have a lot of similarities. They have a finer-grained capability system than is expressible in Rust (or our encoding of it in Section 4) which makes it possible to change the type of a reference cell upon write. 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 4.

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.

However, both Mezzo and ATS have an advantage that we do not: 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 clause programs, much like Prolog 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 [Miller et al. 1987] programs. A generalisation of the Hereditary Harrop fragment to linear logic, described by Hodas and Miller [1994], is the foundation of the Lolli language [Hodas 1994]. The authors also coin the notion of *uniform* proof. A fragment where uniform proofs are complete supports goal-oriented proof search, like Prolog does.

Completeness of uniform proofs is equivalent to Lemma 6.1, which, in turn, is used in the proof of the soundness lemma 6.4. 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 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 do not break our proofs.

Resource usage analysis. Igarashi and Kobayashi [2005] introduce a framework which can be instantiated into many resource analyses (such as proper deallocation of resources) can be modelled and analysed. In particular they give a decision procedure for a wide class of such analyses. Although our objects of study intersect, our purpose is quite different as theirs is a tool for language designer to design analyses, while ours is for programmers to implement new abstractions as libraries.

11 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 to do was to adapt the work of 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 4 is not practical 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

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

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

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

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

J.S. Hodas and D. Miller. 1994. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation* 110, 2 (1994), 327 – 365. https://doi.org/10.1006/inco.1994.1036

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

Atsushi Igarashi and Naoki Kobayashi. 2005. Resource Usage Analysis. *ACM Trans. Program. Lang. Syst.* 27, 2 (March 2005), 264–313. https://doi.org/10.1145/1057387.1057390

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

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

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

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

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

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

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

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

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

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

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

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.

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

Fig. 11. Grammar of the core calculus

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

$$
\text{L-VAR}\ \frac{x{:}_1 \forall \overline{a}.\upsilon \in \Gamma}{\Gamma \vdash x : \upsilon[\overline{\tau}/\overline{a}]}
\qquad
\text{L-ABS}\ \frac{\Gamma, x{:}_\pi \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to_\pi \tau_2}
\qquad
\text{L-APP}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \to_\pi \tau \qquad \Gamma_2 \vdash e_1 : \tau_1}{\Gamma_1 + \pi{\cdot}\Gamma_2 \vdash e_1\ e_2 : \tau}
$$

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

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

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

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 7.1. 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 7 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 $[\![\omega{\cdot}Q]\!]^{\mathrm{ev}}$ into an $\mathbf{Ur}\,([\![Q]\!]^{\mathrm{ev}})$. For any $e : [\![\omega{\cdot}Q]\!]^{\mathrm{ev}}$, we shall write $\underline{e}_Q : \mathbf{Ur}\,([\![\omega{\cdot}Q]\!]^{\mathrm{ev}})$. As a shorthand, particularly useful in nested patterns, we will write

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

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

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

B PROOFS

B.5 Lemmas on the qualified type system

Proof of Lemma 5.4. Let us prove separately the cases $\pi = 1$ and $\pi = \omega$.

- When $\pi = 1$, then $\pi \cdot Q = Q$ for all Q, hence $Q_1 \Vdash Q_2$ implies $\pi \cdot Q_1 \Vdash \pi \cdot Q_2$.
- For the case $\pi = \omega$, let us consider a few properties. First note that, for any Q, $\omega \cdot Q = \omega \cdot Q \otimes \omega \cdot Q$. From which it follows, using the laws of Definition 5.3, that $\omega \cdot Q \Vdash Q_1 \otimes Q_2$ if and only if $\omega \cdot Q \Vdash Q_1$ and $\omega \cdot Q \Vdash Q_2$.
  This means that to verify that $\omega \cdot Q_1 \Vdash \omega \cdot Q_2$, it is equivalent to prove that $\omega \cdot Q_1 \Vdash \omega \cdot q_2$ for each $q_2 \in U$ (letting $\omega \cdot Q_2 = (U, \emptyset)$). In turn, by Definition 5.3 and observing that $\omega \cdot (\omega \cdot Q_1) = Q_1$, this is equivalent to $\omega \cdot Q_1 \Vdash 1 \cdot q_2$.
  This follows from the fact that $Q_1 \Vdash Q_2$ implies $\omega \cdot Q_1 \Vdash Q_2$ (Definition 5.3) and the property, shown above, that $\omega \cdot Q_1 \Vdash Q_2 \otimes Q_2'$ if and only if $\omega \cdot Q_1 \Vdash Q_2$ and $\omega \cdot Q_1 \Vdash Q_2'$.

□

Proof of Lemma 5.5. Let us prove separately the cases $\pi = 1$ and $\pi = \omega$.

- When $\pi = 1$, then $\pi \cdot Q = Q$ for all Q, in particular $Q_1 \Vdash 1 \cdot Q_2$ implies that $Q_1 = 1 \cdot Q_1$ with $Q_1 \Vdash Q_2$.
- When $\pi = \omega$, then let us first remark, letting $\omega \cdot Q_2 = (U, \emptyset)$ that, by a straightforward induction on the cardinality of U it is sufficient to prove that the result holds for atomic constraints.
  That is, we need to prove that if $Q_1 \Vdash \omega \cdot q_2$ then there exists Q' such that $Q_1 = \omega \cdot Q'$ and $Q' \Vdash \rho \cdot q_2$ (for all ρ).
  This result, in turns, holds by Definition 5.3.

□

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

Proof. Immediate by case analysis of π and ρ. □

B.6 Lemmas on constraint inference

Proof of Lemma 6.1. The cases $Q \vdash C_1 \,\&\, C_2$ and $Q \vdash \pi \cdot (Q_2 \Rightarrow\!\circ\, C)$ 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 Definition 5.3.

□

Proof of Lemma 6.2. By induction on the syntax of C

- If $C = Q'$, then the result follows from Lemma 5.4

$$
\left\{
\begin{array}{l}
[\![Q;\Gamma \vdash x : \upsilon\ [\,\overline{\tau}\ /\ \overline{a}\,]]\!]_z = \\
\qquad x\ z \\
[\![Q;\Gamma \vdash \lambda x.e : \tau_1 \to_\pi \tau_2]\!]_z = \\
\quad \lambda x.[\![Q;\Gamma, x :_\pi \tau_1 \vdash e : \tau_2]\!]_z \\
[\![Q_1 \otimes Q_2;\Gamma_1 + \Gamma_2 \vdash e_1\ e_2 : \tau]\!]_z = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, z_2) \to \\
\qquad ([\![Q_1;\Gamma_1 \vdash e_1 : \tau_1 \to_1 \tau]\!]_{z_1})\ ([\![Q_2;\Gamma_2 \vdash e_2 : \tau_1]\!]_{z_2})\} \\
[\![Q_1 \otimes \omega \cdot Q_2;\Gamma_1 + \omega \cdot \Gamma_2 \vdash e_1\ e_2 : \tau]\!]_z = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, \underline{z_2}) \to \\
\qquad ([\![Q_1;\Gamma_1 \vdash e_1 : \tau_1 \to_\omega \tau]\!]_{z_1})\ ([\![Q_2;\Gamma_2 \vdash e_2 : \tau_1]\!]_{z_2})\} \\
[\![Q \otimes Q_1[\overline{\upsilon}/\overline{a}];\Gamma \vdash \mathbf{pack}\ e : \exists\,\overline{a}.\tau \olessthan Q_1]\!]_z = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z', z'') \to \\
\qquad \mathbf{pack}\ (z'', [\![Q;\Gamma \vdash e : \tau[\overline{\upsilon}/\overline{a}]]\!]_{z'})\} \\
[\![Q_1 \otimes Q_2;\Gamma_1 + \Gamma_2 \vdash \mathbf{unpack}\ x = e_1\ \mathbf{in}\ e_2 : \tau]\!]_z = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, z_2) \to \\
\qquad \mathbf{unpack}\ (z', x) = [\![Q_1;\Gamma_1 \vdash e_1 : \exists\,\overline{a}.\tau_1 \olessthan Q]\!]_{z'}\ \mathbf{in} \\
\qquad \mathbf{let}_1\ {z_2}' = (z_2, z')\ \mathbf{in} \\
\qquad [\![Q_2 \otimes Q;\Gamma_2, x :_1 \tau_1 \vdash e_2 : \tau]\!]_{{z_2}'}\} \\
[\![Q_1 \otimes Q_2;\Gamma_1 + \Gamma_2 \vdash \mathbf{let}_1\ x = e_1\ \mathbf{in}\ e_2 : \tau]\!]_z = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, z_2) \to \\
\qquad \mathbf{let}_1\ x : [\![Q]\!]^{\mathbf{ev}} \to_1 \tau_1 = [\![Q_1 \otimes Q;\Gamma_1 \vdash e_1 : \tau_1]\!]_{z_1} \\
\qquad \mathbf{in}\ [\![Q_2;\Gamma_2, x :_1 \tau_1 \vdash e_2 : \tau]\!]_{z_2}\} \\
[\![\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]\!]_z = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(\underline{z_1}, z_2) \to \\
\qquad \mathbf{let}_\omega\ x : [\![Q]\!]^{\mathbf{ev}} \to_1 \tau_1 = [\![Q_1 \otimes Q;\Gamma_1 \vdash e_1 : \tau_1]\!]_{z_1}\ \mathbf{in} \\
\qquad [\![Q_2;\Gamma_2, x :_\omega \tau_1 \vdash e_2 : \tau]\!]_{z_2}\} \\
[\![Q_1 \otimes Q_2;\Gamma_1 + \Gamma_2 \vdash \mathbf{let}_1\ x : \forall\,\overline{a}.Q =\!\!\circ\ \tau_1 = e_1\ \mathbf{in}\ e_2 : \tau]\!]_z = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, z_2) \to \\
\qquad \mathbf{let}_1\ x : \forall\,\overline{a}.[\![Q]\!]^{\mathbf{ev}} \to_1 \tau_1 = [\![Q_1 \otimes Q;\Gamma_1 \vdash e_1 : \tau_1]\!]_{z_1}\ \mathbf{in} \\
\qquad [\![Q_2;\Gamma_2, x :_1 \forall\,\overline{a}.Q =\!\!\circ\ \tau_1 \vdash e_2 : \tau]\!]_{z_2}\} \\
[\![\omega \cdot Q_1 \otimes Q_2;\omega \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}_\omega\ x : \forall\,\overline{a}.Q =\!\!\circ\ \tau_1 = e_1\ \mathbf{in}\ e_2 : \tau]\!]_z = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(\underline{z_1}, z_2) \to \\
\qquad \mathbf{let}_\omega\ x : \forall\,\overline{a}.[\![Q]\!]^{\mathbf{ev}} \to_1 \tau_1 = [\![Q_1 \otimes Q;\Gamma_1 \vdash e_1 : \tau_1]\!]_{z_1}\ \mathbf{in} \\
\qquad [\![Q_2;\Gamma_2, x :_\omega \tau_1 \vdash e_2 : \tau]\!]_{z_2}\} \\
[\![\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]\!]_z = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(\underline{z_1}, z_2) \to \\
\qquad \mathbf{case}_1\ ([\![Q_1;\Gamma_1 \vdash e : T\ \overline{\tau}]\!]_{z_1})\ \mathbf{of} \\
\qquad\quad \{\overline{K\ \overline{x}_i \to [\![Q_2;\Gamma_2, \overline{x_i :_{(\pi \cdot \pi_i)} \upsilon_i[\overline{\tau}/\overline{a}]} \vdash e_i : \tau]\!]_{z_2}}\}\} \\
[\![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]\!]_z = \\
\quad \mathbf{case}_1\ z\ \mathbf{of}\ \{(z_1, z_2) \to \\
\qquad \mathbf{case}_\omega\ ([\![Q_1;\Gamma_1 \vdash e : T\ \overline{\tau}]\!]_{z_1})\ \mathbf{of} \\
\qquad\quad \{\overline{K\ \overline{x}_i \to [\![Q_2;\Gamma_2, \overline{x_i :_{(\pi \cdot \pi_i)} \upsilon_i[\overline{\tau}/\overline{a}]} \vdash e_i : \tau]\!]_{z_2}}\}\}
\end{array}
\right.
$$

Fig. 13. Desugaring

- If $C = C_1 \otimes C_2$, then we can prove the result like we proved the corresponding case in Lemma 5.4, using Lemma 6.1.
- 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 6.1, 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 6.1, 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 6.3. By induction on the syntax of C

- If $C = Q'$, then the result follows from Lemma 5.5
- If $C = C_1 \otimes C_2$, then we can prove the result like we proved the corresponding case in Lemma 5.5 using Lemma 6.1.
- 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 6.1, 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 6.1.

□

Proof of Lemma 6.4. By induction on $\Gamma \mapsto e : \tau \rightsquigarrow C$

G-Var We have
- $\Gamma_1 = x{:}_1\forall\overline{a}.Q \multimap v$
- $\Gamma_1 + \omega{\cdot}\Gamma_2 \mapsto x : v[\overline{\tau}/\overline{a}] \rightsquigarrow Q[\overline{\tau}/\overline{a}]$
- $Q_g \vdash Q[\overline{\tau}/\overline{a}]$

Therefore, by rules E-Var and E-Sub, it follows immediately that $Q_g; \Gamma_1 + \omega{\cdot}\Gamma_2 \vdash x : v[\overline{\tau}/\overline{a}]$

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

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

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

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

By Lemmas 6.1 and 6.3, 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 $Q_g; \pi{\cdot}\Gamma_1 + \Gamma_2 \vdash \mathbf{let}_\pi\ x = e_1\ \mathbf{in}\ e_2 : \tau$.

G-LetSig We have

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

By Lemmas 6.1 and 6.3, 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

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

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

G-App We have

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

By Lemmas 6.1 and 6.3, 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 $Q_g; \Gamma_1 + \pi{\cdot}\Gamma_2 \vdash e_1\ e_2 : \tau$.

G-Pack We have

- $\Gamma \mapsto \mathbf{pack}\ e : \exists\overline{a}.\tau \owedge Q \rightsquigarrow C \otimes Q[\overline{v}/\overline{a}]$
- $Q_g \vdash C \otimes Q[\overline{v}/\overline{a}]$
- $\Gamma \mapsto e : \tau[\overline{v}/\overline{a}] \rightsquigarrow C$

By Lemma 6.1, there exist Q_1, Q_2 such that

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

By induction hypothesis

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

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

G-Unpack We have

- $\Gamma_1 + \Gamma_2 \mapsto \mathbf{unpack}\ x = e_1\ \mathbf{in}\ e_2 : \tau \rightsquigarrow C_1 \otimes 1{\cdot}(Q' \multimap C_2)$
- $Q_g \vdash C_1 \otimes 1{\cdot}(Q' \multimap C_2)$
- $\Gamma_1 \mapsto e_1 : \exists\overline{a}.\tau_1 \owedge Q' \rightsquigarrow C_1$
- $\Gamma_2, x{:}_\pi \tau_1 \mapsto e_2 : \tau \rightsquigarrow C_2$

By Lemma 6.1, 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

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

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

G-Case We have

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

By repeated uses of Lemma 6.1 as well as Lemma 6.3, 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 $Q_g; \pi \cdot \Gamma + \Delta \vdash \textbf{case}_\pi\ e\ \textbf{of}\ \{\overline{K_i\ \overline{x_i} \longrightarrow e_i}\} : \tau$.

□

Proof of Lemma 6.5. By induction on $U; L_i \vdash_{\mathsf{s}} C \rightsquigarrow L_o$

S-Atom We have

- $U; L_i \vdash_{\mathsf{s}} \pi \cdot q \rightsquigarrow L_o$
- $U; L_i \vdash_{\mathsf{s}}^{\mathsf{atom}} \pi \cdot q \rightsquigarrow L_o$

By Property 6.6 we have

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

Then by C-Dom we have $(U, L_i) \vdash \pi \cdot q \otimes (\emptyset, L_o)$.

S-Add We have

- $U; L_i \vdash_{\mathsf{s}} C_1 \& C_2 \rightsquigarrow L_o$
- $U; L_i \vdash_{\mathsf{s}} C_1 \rightsquigarrow L_o$
- $U; L_i \vdash_{\mathsf{s}} C_2 \rightsquigarrow L_o$

By induction hypothesis we have

- $L_o \subseteq L_i$
- $(U, L_i) \vdash C_1 \otimes (\emptyset, L_o)$
- $(U, L_i) \vdash C_2 \otimes (\emptyset, L_o)$

Then by C-With we have $(U, L_i) \vdash C_1 \& C_2 \otimes (\emptyset, L_o)$.

S-Mult We have

- $U; L_i \vdash_{\mathsf{s}} C_1 \otimes C_2 \rightsquigarrow L_o$
- $U; L_i \vdash_{\mathsf{s}} C_1 \rightsquigarrow L_o'$
- $U; L_o' \vdash_{\mathsf{s}} C_2 \rightsquigarrow L_o$

By induction hypothesis we have

- $L_o \subseteq L_o'$
- $L_o' \subseteq L_i$
- $(U, L_i) \vdash C_1 \otimes (\emptyset, L_o')$
- $(U, L_o') \vdash C_2 \otimes (\emptyset, L_o)$

Then by transitivity of $\subseteq$ we have $L_o \subseteq L_i$, and by C-Tensor we have $(U, L_i) \otimes (U, L'_o) \vdash C_1 \otimes C_2 \otimes (\emptyset, L'_o) \otimes (\emptyset, L_o)$ by Lemma 6.1 we have $(U, L_i) \vdash C_1 \otimes C_2 \otimes (\emptyset, L_o)$.

S-ImplOne We have

- $U; L_i \vdash_s 1 \cdot ((U_0, L_0) \multimap C) \rightsquigarrow L_o$
- $U \cup U_0; L_i \uplus L_0 \vdash_s C \rightsquigarrow L_o$
- $L_o \subseteq L_i$

By induction hypothesis we have

- $(U \cup U_0, L_i \uplus L_0) \vdash C \otimes (\emptyset, L_o)$
- $L_o \subseteq L_i \uplus L_0$

Then we know that $(\emptyset, L_i) = (\emptyset, L_o) \otimes (\emptyset, L'_i)$ for some L'_i. Then by Lemma 6.1 we know that $(U \cup U_0, L'_i \uplus L_0) \vdash C$ and by C-Impl we have $(U, L'_i) \vdash 1 \cdot ((U_0, L_0) \multimap C)$. Finally, by C-Tensor we conclude that $(U, L_i) \vdash 1 \cdot ((U_0, L_0) \multimap C) \otimes (\emptyset, L_o)$

S-ImplMany We have

- $U; L_i \vdash_s \omega \cdot ((U_0, L_0) \multimap C) \rightsquigarrow L_i$
- $U \cup U_0; L_0 \vdash_s C \rightsquigarrow \emptyset$

By induction hypothesis we have

- $(U \cup U_0, L_0) \vdash C \otimes (\emptyset, \emptyset)$

Then by Lemma 6.1 we have $(U \cup U_0, L_0) \vdash C$ and by C-Impl $(U, \emptyset) \vdash \omega \cdot ((U_0, L_0) \multimap C)$ and finally by rule C-Tensor we have $(U, L_i) \vdash \omega \cdot ((U_0, L_0) \multimap C) \otimes (\emptyset, L_i)$. $L_i \subseteq L_i$ holds trivially. □

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 5.3, 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 $C = Q$. □