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

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

Additional Key Words and Phrases: GHC, Haskell, 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. Rust uses dedicated syntax and code generation to support management of resources, while Linear Haskell is a type system change without any other support from the compiler. Linear Haskell is designed to be general purpose, but using linear arguments to emulate Rust's ownership model is a painful exercise, 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 function:

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

This function reads the first two elements of an array and returns them after deallocating the array. Linearity enables the array library to ensure that there is only one reference to the array, and therefore it can be mutated in-place without violating referential transparency. After the array has been freed, it is no longer possible to read or write to it. Notice that the *read* function consumes the

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.

2022. 2475-1421/2022/1-ART1 $15.00
https://doi.org/

array and returns a fresh array, to be used in future operations. Operationally, the array remains the same, but each operation assigns a new name to it, thus facilitating tracking references statically. Finally, *free* consumes the array without returning a new one, statically guaranteeing that it can no longer be used.[1] The values x and y read from the array are returned; their types include elements wrapped by the *Ur* (pronounced "unrestricted") type, allowing them to be used arbitrarily many times. This works because *read2AndDiscard* takes a restricted-use array containing unrestricted elements. In a non-linear language, one would have to forgo referential transparency to handle mutable operations either by using a monadic interface or allowing arbitrary effects. Compare the above function with what one would write in a non-linear, impure language:

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

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

Rust introduces the *borrow checker* for this very purpose. Our approach is, in some ways, more lightweight: 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 arrays or file handles. Thus, linear constraints are the combination of these two concepts, which have been studied independently elsewhere [Bernardy et al. 2017; Cervesato et al. 2000; Hodas and Miller 1994; Vytiniotis et al. 2011].

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

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

The only changes from the impure version are that this version explicitly requires having read and write access to the array, and explicit **pack** annotations are used to indicate the binders that require special treatment (Section 8.3 suggests how we can get rid of the **pack**, too). Crucially, the resource representing the ownership of the array is a linear constraint and is separate from the array itself, which no longer needs to be threaded manually.

Our contributions are as follows:

[1] We assume that all pattern bindings in **let**-expressions are *strict*. This ensures that the array actually gets freed. We discuss this point in Section 2.

99
100
101
102
103
104
105
106
107
108
109
110
111
112
113

- 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 points towards a possible unification of 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.

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 $a \multimap b$.[2] 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*) 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:

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

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

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

From the perspective of the programmer, this is unwanted boilerplate. The approach with linear constraints is to let the array behave non-linearly, and let its capabilities (i.e., having read or write

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

114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147

$$
\begin{aligned}
&\mathit{new} :: \mathit{Int} \to (\mathit{MArray}\ a \multimap \mathit{Ur}\ r) \multimap \mathit{Ur}\ r\\
&\mathit{write} :: \mathit{MArray}\ a \multimap \mathit{Int} \to a \to \mathit{MArray}\ a\\
&\mathit{read} :: \mathit{MArray}\ a \multimap \mathit{Int} \to (\mathit{MArray}\ a, \mathit{Ur}\ a)\\
&\mathit{free} :: \mathit{MArray}\ a \multimap ()
\end{aligned}
$$

(a) Linear Types

$$
\begin{aligned}
&\mathbf{type}\ \mathit{RW}\ n = (\mathit{Read}\ n, \mathit{Write}\ n)\\
&\mathit{new} :: \mathit{Int} \to (\forall\ n.\ \mathit{RW}\ n \mathrel{=\!\!\circ} \mathit{UArray}\ a\ n \to \mathit{Ur}\ r) \multimap \mathit{Ur}\ r\\
&\mathit{write} :: \mathit{RW}\ n \mathrel{=\!\!\circ} \mathit{UArray}\ a\ n \to \mathit{Int} \to a \to () \otimes \mathit{RW}\ n\\
&\mathit{read} :: \mathit{Read}\ n \mathrel{=\!\!\circ} \mathit{UArray}\ a\ n \to \mathit{Int} \to \mathit{Ur}\ a \otimes \mathit{Read}\ n\\
&\mathit{free} :: \mathit{RW}\ n \mathrel{=\!\!\circ} \mathit{UArray}\ a\ n \to ()
\end{aligned}
$$

(b) Linear Constraints

Fig. 1. Interfaces for mutable arrays

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

In Section 1 and in the rest of the paper, we use pattern matches in **let** bindings. By default in Haskell, patterns in **let**s are lazy, which means that **let** $(a, b) = p$ **in** $()$ will not actually evaluate the pair. To force evaluation, a strictness annotation can be added: **let** $!(a, b) = p$ **in** $()$. Pattern matching in linear **let** bindings must always be strict in Linear Haskell, so writing the lazy version would be rejected by the compiler. To simplify the presentation, we assume that all patterns in **let** bindings are strict. Strict **let** pattern bindings can be desugared into **case** expressions: **case** p **of** $(a, b) \to ()$.

3 WORKING WITH LINEAR CONSTRAINTS

Consider the Haskell function *show*:

$$\mathit{show} :: \mathit{Show}\ a \Rightarrow a \to \mathit{String}$$

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

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

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

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

There are a few things to notice:

- We have introduced a new type variable n. In contrast, the version in Figure 1a without linear constraints has type $\mathit{free} :: \mathit{MArray}\ a \multimap ()$. The type variable n is a type-level tag used to identify the array. Ideally, the linear constraint would refer directly to the array value, and have the dependent type $\mathit{free} :: (n :: \mathit{Array}\ a) \to \mathit{RW}\ n \mathrel{=\!\!\circ} ()$. While giving a compile-time name to a function argument is common in dependently typed languages such as ATS [Xi 2017] or Idris, our approach, on the other hand, shows how we can still link a run-time value and a compile-time tag without needing any dependent types.

197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245

- The run-time variable representing the array can now be used multiple times. Instead of restricting the use of this variable, the linear constraint $RW\ n$ controls access to the array.
- If we have a single, linear, $RW\ n$ available, then after *free* there will not be any $RW\ n$ left to use, thus preventing the array from being used after freeing. This is precisely what we were trying to achieve.

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

$$new :: Int \rightarrow (\forall\ n.\ RW\ n \multimap UArray\ a\ n \rightarrow Ur\ r) \multimap Ur\ r$$

This higher-rank function can be thought of as a computation that allocates an array in a *scope*. The scope is given an array with read and write capability. The scope must return an unrestricted value to ensure that the linear constraint cannot be embedded into the return value. The construction here ensures that, within the scope, we can be sure both that there is one unique owner of the array at all times, and that the array is freed at the end, given that the only way to remove the $RW\ n$ constraint is to use *free*.

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

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

This type has a new symbol, $\otimes$, which allows us to pack a produced constraint with a returned value. We will see in Section 4.2 that these produced constraints will also sometimes need to come with fresh type variables. Combining these ideas, we introduce[3] a type construction $\exists\ a_1\ \dots\ a_n.t \otimes Q$, where Q is a linear constraint (with the $a_1\ \dots\ a_n$ in scope) that is paired with the type t.[4] Such types are introduced with the **pack** constructor. When pattern-matching on a **pack** constructor, all existentially quantified type variables are brought into scope and all the packed constraints are assumed. We have now seen all the ingredients needed to write the *read2AndDiscard* example as in Section 1.

3.1 Minimal Examples

To get a sense of how the features 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 =o Bool -> Int
dithering x = if x then useC else 10
```

[3] 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 generalisation of those presented by Pierce [2002, Chapter 24]. However, a recent publication by Eisenberg et al. [2021] works out an approach that will make linear constraints easier to use, as we discuss in Section 8.3.

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

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

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

$$\mathit{const} :: a \multimap b \to a$$

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:

$$\mathit{neglecting} :: C \Rightarrow\!\!\circ\ \mathit{Int}$$
$$\mathit{neglecting} = \mathit{const}\ 10\ \mathit{useC}$$

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,

$$\mathit{notNeglecting} :: C \Rightarrow\!\!\circ\ \mathit{Int}$$
$$\mathit{notNeglecting} = \mathit{const}\ \mathit{useC}\ 10$$

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

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

$$\mathit{overusing} :: C \Rightarrow\!\!\circ\ (\mathit{Int}, \mathit{Int})$$
$$\mathit{overusing} = (\mathit{useC}, \mathit{useC})$$

4 APPLICATION: MEMORY OWNERSHIP

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

4.1 Capability constraints

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

$$\textbf{class}\ \mathit{Read}\ (n :: \mathit{Location}) \qquad\qquad \textbf{class}\ \mathit{Write}\ (n :: \mathit{Location})$$

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

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

$$\textbf{data}\ AtomRef\ (a :: Type)\ (n :: Location)$$

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. As with *new*, the return value must be unrestricted.

$$newRef :: (\forall\ n.\ RW\ n \Rightarrow\!\!\circ\ AtomRef\ a\ n \to Ur\ b) \multimap Ur\ b$$

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

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

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

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

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

$$freeRef :: RW\ n \Rightarrow\!\!\circ\ AtomRef\ a\ n \to ()$$

4.2 Arrays

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

$$\textbf{data}\ PArray\ (a :: Location \to Type)\ (n :: Location)$$
$$newPArray :: Int \to (\forall\ n.\ RW\ n \Rightarrow\!\!\circ\ PArray\ a\ n \to Ur\ b) \multimap Ur\ b$$

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

As discussed in Section 3, the scope of *newPArray* returns an unrestricted value to ensure that the linear constraint is consumed within the scope (since linear values cannot be embedded into an unrestricted value). As this is the only introduction form of $RW\ n$, it can safely be assumed to be unique within the scope. An alternative design would be to require that the scope returns the linear constraint: $\forall\ n. RW\ n \Rightarrow\!\!\circ\ PArray\ a\ n \to b \otimes RW\ n$. This version is less flexible, because it doesn't allow the scope to deallocate or freeze the array.

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

When writing a reference (be it an array or an *AtomRef*) in an array, ownership of the reference is transferred to the array. Because the content of an array cell is linear, we can't erase it when we write, instead we get the old content back. To do so, we use the *exchange* primitive

$$exchange :: (RW\ n, RW\ p) \Rrightarrow a\ p \rightarrow a\ n \rightarrow () \otimes (RW\ n, RW\ p)$$

which exchanges two references, together with the borrowing primitives described below.

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

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

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

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

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

$$read :: Read\ n \Rrightarrow UArray\ a\ n \rightarrow Int \rightarrow Ur\ a \otimes Read\ n$$
$$read\ arr\ i = lend\ arr\ i\ readRef$$

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

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

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

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

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

Note that the constraint $Ur\ (Slices\ n\ l\ r)$ is unrestricted, which means that it is not *necessary* to join the two arrays before deallocating them: they can be deallocated separately.

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

```
swap :: RW n =o PArray a n → Int → Int → () ⧀ RW n
swap arr i j | i ≡ j = pack ()
             | i > j = swap arr j i
             | i < j = let pack (Ur (l, r)) = split arr (i + 1)
                           pack ()         = lendMut l i (λaᵢ →
                                                 let pack () = lendMut r (j − (i + 1)) (λaⱼ →
                                                   let pack () = exchange aᵢ aⱼ in pack ()) in pack ())
                           pack (Ur _)     = join l r
                       in pack ()
```

Fig. 2. Swapping two elements of an array

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

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

Fig. 3. In-place quicksort

4.2.3 In-place quicksort. As an example of using the machinery defined above, we implement an in-place, pure quicksort algorithm, given in Figure 3. The *partition* function is responsible for picking a pivot element and reorganising the array elements such that each element preceding the pivot will be less than or equal to it, and the elements after will be greater than the pivot. Once finished, it returns the index of the pivot element; *sort* then splits the array at the pivot element and recursively operates on the two slices. These recursive calls can be executed non-deterministically (or, indeed, in parallel), as the type system guarantees that no data races can occur.

5 A QUALIFIED TYPE SYSTEM FOR LINEAR CONSTRAINTS

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

5.1 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

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

442 them in blue like constraints. For our purposes, we need only the simplest system of multiplicity,
443 comprising 1 (representing linear functions) and ω (representing regular Haskell functions).
444
445 $$\pi, \rho \quad ::= \quad 1 \mid \omega \quad \text{Multiplicities}$$
446
447 The idea of multiplicity goes back at least to Kobayashi et al. [1999]. The power of multiplicities is
448 that they can encode the structural rules of linear logic with only the semiring operations: addition
449 and multiplication. Here and in the rest of the paper we adopt the convention that equations
450 defining a function by pattern matching are marked with a { to their left.
451
452 $$\left\{ \pi + \rho = \omega \right. \qquad \left\{ \begin{array}{lcl} 1 \cdot \pi & = & \pi \\ \omega \cdot \pi & = & \omega \end{array} \right.$$
453
454 Even though linear Haskell additionally supports multiplicity polymorphism, we do not support
455 multiplicity polymorphism on constraint arguments. Multiplicity polymorphism of regular func-
456 tion arguments is used to avoid duplicating the definition of higher-order functions. The prototyp-
457 ical example is $map :: (a \to_m b) \to [a] \to_m [b]$, where $\to_m$ is the notation for a function arrow
458 of multiplicity m. First-order functions, on the other hand, do not need multiplicity polymorphism,
459 because linear functions can be η-expanded into unrestricted function as explained in Section 2.
460 Higher-order functions whose arguments are themselves constrained functions are rare, so we do
461 not yet see the need to extend multiplicity polymorphism to apply to constraints. Futhermore, it
462 is not clear how to extend the constraint solver of Section 6.3 to support multiplicity-polymorphic
463 constraints.
464
465

5.2 Simple Constraints and Entailment

466 Let us now turn to constraints themselves. We call constraints such as *Read n* or *Write n* *atomic*
467 *constraints*. The exact nature of atomic constraints is left unspecified: the set of atomic constraints
468 is a parameter of our qualified type system.
469
470 *Definition 5.1 (Atomic constraints).* The qualified type system is parameterised by a set, whose
471 elements are called *atomic constraints*. We use the variable q to denote atomic constraints.
472
473 Atomic constraints are assembled into *simple constraints* Q, which play the hybrid role of con-
474 straint contexts and (linear) logic formulae. The following operations work with simple constraints:
475
476 **Scaled atomic constraints** $\pi \cdot q$ is a simple constraint, where π specifies whether q is to be used linearly or not.
477 **Conjunction** Two simple constraints can be paired up $Q_1 \otimes Q_2$. Semantically, this corresponds
478 to the multiplicative conjunction of linear logic. Tensor products represent pairs of con-
479 straints such as *(Read n, Write n)* from Haskell.
480 **Empty conjunction** Finally we need a neutral element ε to the tensor product. The empty
481 conjunction is used to represent functions which don't require any constraints.
482
483 However, we do not define Q inductively, because we require certain equalities to hold:
484
485 $$\begin{array}{rcl} Q_1 \otimes Q_2 & = & Q_2 \otimes Q_1 \\ (Q_1 \otimes Q_2) \otimes Q_3 & = & Q_1 \otimes (Q_2 \otimes Q_3) \end{array} \qquad \begin{array}{rcl} \omega \cdot q \otimes \omega \cdot q & = & \omega \cdot q \\ Q \otimes \varepsilon & = & Q \end{array}$$
486
487 We thus say that a simple constraint is a pair combining a set of unrestricted constraints U
488 and a multiset of linear constraints L. The linear constraints must be stored in a multiset, because
489 assuming the same constraint twice is distinct from assuming it only once.
490

491 $Q \Vdash Q$
492 if $Q_1 \Vdash Q_2$ and $Q \otimes Q_2 \Vdash Q_3$ then $Q \otimes Q_1 \Vdash Q_3$
493 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 $Q'' \Vdash Q_2$
494 if $Q \Vdash \varepsilon$ then there exists Q' such that $Q = \omega \cdot Q'$
495 if $Q_1 \Vdash Q_1'$ and $Q_2 \Vdash Q_2'$ then $Q_1 \otimes Q_2 \Vdash Q_1' \otimes Q_2'$
496 if $Q \Vdash \rho \cdot q$ then $\pi \cdot Q \Vdash (\pi \cdot \rho) \cdot q$
497 if $Q \Vdash (\pi \cdot \rho) \cdot q$ then there exists Q' such that $Q = \pi \cdot Q'$ and $Q' \Vdash \rho \cdot q$
498 if $Q_1 \Vdash Q_2$ then $\omega \cdot Q_1 \Vdash Q_2$
499 if $Q_1 \Vdash Q_2$ then for all $Q', \omega \cdot Q' \otimes Q_1 \Vdash Q_2$
500
501 Fig. 4. Requirements for the entailment relation $Q_1 \Vdash Q_2$
502
503 *Definition 5.2 (Simple constraints).*
504
505 $$U ::= \ldots \quad \text{set of atomic constraints } q$$
506 $$L ::= \ldots \quad \text{multiset of atomic constraints } q$$
507 $$Q ::= (U, L) \quad \text{simple constraints}$$
508 We can now straightforwardly define the operations we need on simple constraints:
509
510 $$\varepsilon = (\emptyset, \emptyset) \qquad \begin{cases} 1 \cdot q = (\emptyset, q) \\ \omega \cdot q = (q, \emptyset) \end{cases} \qquad (U_1, L_1) \otimes (U_2, L_2) = (U_1 \cup U_2, L_1 \uplus L_2)$$
511
512 In practice, we do not need to concern ourselves with the concrete representation of Q as a pair
513 of sets, instead using the operations defined just above.
514 The semantics of simple constraints (and, indeed, of atomic constraints) is given by an *entailment*
515 *relation*. Just like the set of atomic constraints, the entailment relation is a parameter of our system
516
517 *Definition 5.3 (Entailment relation).* The qualified type system is parameterised by a relation
518 $Q_1 \Vdash Q_2$ between two simple constraints. The entailment relation must obey the laws listed in
519 Figure 4.
520
521 An important feature of simple constraints is that, while scaling syntactically happens at the
522 level of atomic constraints, these properties of scaling extend to scaling of arbitrary constraints.
523 Define $\pi \cdot Q$ as:
524
525 $$\begin{cases} 1 \cdot (U, L) = (U, L) \\ \omega \cdot (U, L) = (U \cup L, \emptyset) \end{cases}$$
526 Then the following properties hold
527
528 Lemma 5.4 (Scaling). *If $Q_1 \Vdash Q_2$, then $\pi \cdot Q_1 \Vdash \pi \cdot Q_2$.*
529
530 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'.*
531
532 Corollary 5.6 (Linear assumptions). *If $Q_1 \Vdash \omega \cdot Q_2$, then Q_1 contains no linear assumptions.*
533
534 Proofs of these lemmas (and others) appear in our anonymised supplementary material; they
535 can be proved by straightforward use of the properties in Figure 4.

5.3 Typing rules

536 With this material in place, we can now present our type system. The grammar is given in Figure 5,
537 which also includes the definitions of scaling on contexts $\pi \cdot \Gamma$ and addition of contexts $\Gamma_1 + \Gamma_2$. Note
538 that addition on contexts is actually a partial function, as it requires that, if a variable x is bound in
539

a, b Type vars $\quad x, y$ Expression vars $\quad T$ Type constructors $\quad K$ Data constructors

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

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

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

Fig. 5. Grammar of the qualified type system

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

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

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

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

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

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

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

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

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

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

Fig. 6. Qualified type system

both Γ_1 and Γ_2, then x is assigned the same type in both (but perhaps different multiplicities). This partiality is not a problem in practice, as the required condition for combining contexts is always satisfied.

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

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

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

Linear functions. The type of linear functions is written $a \to_1 b$. Despite our focus on linear constraints, we still need linearity in ordinary arguments. Indeed, the linearity of arrows interacts in interesting ways with linear constraints: If $f : a \to_\omega b$ and $x : 1{\cdot}q =\!\!\circ\, a$, then calling $f\ x$ would actually use q many times. We must make sure it is impossible to derive $1{\cdot}q; f{:}_\omega a \to_\omega b, x{:}_\omega 1{\cdot}q =\!\!\circ\, a \vdash f\ x : b$. Otherwise we could make, for instance, the *overusing* function from Section 3.1.3. You can check that $1{\cdot}q; f{:}_\omega a \to_\omega b, x{:}_\omega 1{\cdot}q =\!\!\circ\, a \vdash f\ x : b$ 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 $1{\cdot}q; f{:}_\omega a \to_1 b, x{:}_\omega 1{\cdot}q =\!\!\circ\, a \vdash f\ x : b$ when f is a linear function.

Variables. As is standard, the rule E-Var rule works in a context containing more than just the used binding for x. However, crucially, our rule allows only *unrestricted* variables to be discarded; linear variables *must* be used. We can see this in the rule by noticing that the context has an unrestricted component $\omega{\cdot}\Gamma_2$. The Γ_1 component might be restricted or might not, allowing this rule to apply both for restricted and unrestricted x.

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

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

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

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

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

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

6.1 Wanted constraints

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

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

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 simple constraints in the simple-constraint case:

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

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

687 $\boxed{Q \vdash C}$ *(Wanted-constraint entailment)*

688–691

$$\text{C-DOM}\ \frac{Q_1 \Vdash Q_2}{Q_1 \vdash Q_2} \qquad \text{C-TENSOR}\ \frac{Q_1 \vdash C_1 \qquad Q_2 \vdash C_2}{Q_1 \otimes Q_2 \vdash C_1 \otimes C_2} \qquad \text{C-WITH}\ \frac{Q \vdash C_1 \qquad Q \vdash C_2}{Q \vdash C_1 \,\&\, C_2} \qquad \text{C-IMPL}\ \frac{Q_0 \otimes Q_1 \vdash C}{\pi{\cdot}Q_0 \vdash \pi{\cdot}(Q_1 \multimap C)}$$

692

693 Fig. 7. Wanted-constraint entailment

694

695 We define an entailment relation over wanteds in Figure 7. Note that this relation uses only
696 simple constraints Q as assumptions, as there is no way to assume the more elaborate C[6].
697 Before we move on to constraint generation proper, let us highlight a few technical, yet essential,
698 lemmas about the wanted-constraint entailment relation.

699

700 LEMMA 6.1 (INVERSION). *The inference rules of $Q \vdash C$ can be read bottom-up as well as top-down,*
701 *as is required of $Q_1 \Vdash Q_2$ in Figure 4. That is:*

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

705 LEMMA 6.2 (SCALING). *If $Q \vdash C$, then $\pi{\cdot}Q \vdash \pi{\cdot}C$.*

706

707 LEMMA 6.3 (INVERSION OF SCALING). *If $Q \vdash \pi{\cdot}C$ then $Q' \vdash C$ and $Q = \pi{\cdot}Q'$ for some Q'.*

708

709 ## 6.2 Constraint generation

710 The process of inferring constraints is split into two parts: generating constraints, which we do in
711 this section, then solving them in Section 6.3. Constraint generation is described by the judgement
712 $\Gamma \mapsto e : \tau \rightsquigarrow C$ (defined in Figure 8) which outputs a constraint C required to make e typecheck.
713 The definition $\Gamma \mapsto e : \tau \rightsquigarrow C$ is syntax directed, so it can directly be read as an algorithm, taking as
714 input a *typing derivation* for $\Gamma \vdash e : \tau$ (produced by an external type inference oracle as discussed
715 above). Notably, the algorithm has access to the context splitting from the (previously computed)
716 typing derivation, and is thus indeed syntax directed.
717 The rules of Figure 8 constitute a mostly unsurprising translation of the rules of Figure 6, except
718 for the following points of interest:
719 *Case expressions.* Note the use of $\&$ in the conclusion of rule G-CASE. We require that each branch
720 of a **case** expression use the exact same (linear) assumptions; this is enforced by combining the
721 emitted constraints with $\&$, not $\otimes$. This can also be understood in terms of the array example of
722 Section 1: if an array is freed in one branch of a **case**, we require it to be freed (or freezed) in the
723 other branches too. Otherwise, the array's state will be unknown to the type system after the **case**.
724 *Implications.* The introduction of constraints local to a definition (rule G-LETSIG) corresponds
725 to emitting an implication constraint.
726 *Unannotated* **let**. However, the G-LET rule does not produce an implication constraint, as we do
727 not model **let**-generalisation [Vytiniotis et al. 2010].

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

731

732 LEMMA 6.4 (SOUNDNESS OF CONSTRAINT GENERATION). *For all Q_g, if $\Gamma \mapsto e : \tau \rightsquigarrow C$ and $Q_g \vdash C$*
733 *then $Q_g; \Gamma \vdash e : \tau$.*

734 [6]Allowing the full wanted-constraint syntax in assumptions is the subject of work by Bottu et al. [2017].
735

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

G-VAR
$$\frac{\Gamma_1 = x{:}_1 \forall \overline{a}. Q \Rightarrow\!\!\circ\; \upsilon}{\Gamma_1 + \omega{\cdot}\Gamma_2 \Vdash x : \upsilon[\overline{\tau}/\overline{a}] \rightsquigarrow Q[\overline{\tau}/\overline{a}]}$$

G-ABS
$$\frac{\Gamma, x{:}_\pi \tau_0 \Vdash e : \tau \rightsquigarrow C}{\Gamma \Vdash \lambda x.e : \tau_0 \rightarrow_\pi \tau \rightsquigarrow C}$$

G-APP
$$\frac{\Gamma_1 \Vdash e_1 : \tau_2 \rightarrow_\pi \tau \rightsquigarrow C_1 \qquad \Gamma_2 \Vdash e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma_1 + \pi{\cdot}\Gamma_2 \Vdash e_1\, e_2 : \tau \rightsquigarrow C_1 \otimes \pi{\cdot}C_2}$$

G-PACK
$$\frac{\Gamma \Vdash e : \tau[\overline{\upsilon}/\overline{a}] \rightsquigarrow C}{\Gamma \Vdash \mathbf{pack}\, e : \exists \overline{a}.\tau \otimes Q \rightsquigarrow C \otimes Q[\overline{\upsilon}/\overline{a}]}$$

G-UNPACK
$$\frac{\Gamma_1 \Vdash e_1 : \exists \overline{a}.\tau_1 \otimes Q_1 \rightsquigarrow C_1 \qquad \overline{a} \text{ fresh} \qquad \Gamma_2, x{:}_1 \tau_1 \Vdash e_2 : \tau \rightsquigarrow C_2}{\Gamma_1 + \Gamma_2 \Vdash \mathbf{unpack}\, x = e_1 \,\mathbf{in}\, e_2 : \tau \rightsquigarrow C_1 \otimes 1{\cdot}(Q_1 \Rightarrow\!\!\circ\; C_2)}$$

G-CASE
$$\frac{\Gamma \Vdash e : T\,\overline{\sigma} \rightsquigarrow C \qquad \overline{K_i : \forall \overline{a}.\overline{\upsilon_i} \rightarrow_{\overline{\pi_i}} T\,\overline{a} \qquad \Delta, \overline{x_i{:}_{(\pi\cdot\pi_i)} \upsilon_i[\overline{\sigma}/\overline{a}]} \Vdash e_i : \tau \rightsquigarrow C_i}}{\pi{\cdot}\Gamma + \Delta \Vdash \mathbf{case}_\pi\, e\, \mathbf{of}\, \{\overline{K_i\, \overline{x_i} \rightarrow e_i}\} : \tau \rightsquigarrow \pi{\cdot}C \otimes \&\, C_i}$$

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

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

Fig. 8. Constraint generation

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.

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:

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

785 $\boxed{U; L_i \vdash_\mathsf{s} C_w \rightsquigarrow L_o}$ *(Constraint solving)*

786
787
788
789
790
791
792
793
794

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

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

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

795 Fig. 9. Constraint solver

796
797
798
799 It has a similar structure to the main solver, but only deals with atomic constraints. Even though
800 the main solver is parameterised by this atomic-constraint solver, we will give an instantiation in
801 Section 6.3.2. We require the following property of the atomic-constraint solver:

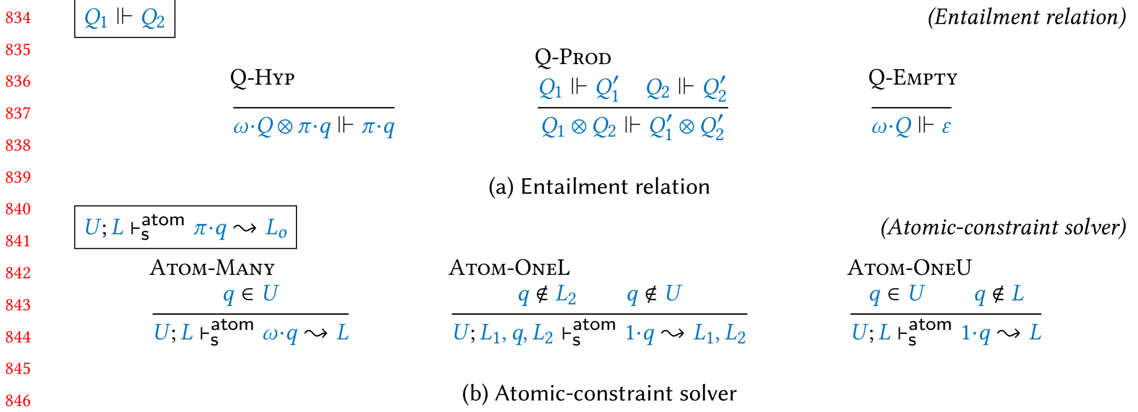
802
803 Property 6.6 (Atomic-constraint solver soundness). *If* $U; L_i \vdash_\mathsf{s}^{\mathsf{atom}} \pi{\cdot}q \rightsquigarrow L_o$*, then:*

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

806
807 *6.3.1 Constraint solver algorithm.* Building on this atomic-constraint solver, we use a linear proof
808 search algorithm based on the recipe given by Cervesato et al. [2000]. Figure 9 presents the rules
809 of the constraint solver.

810 - The S-Mult rule proceeds by solving one side of a conjunction first, then passing the output
811 constraints to the other side. The unrestricted context is shared between both sides.
812 - The S-Add rule handles additive conjunction. The linear constraints are shared between the
813 branches (since additive conjunction is generated from **case** expressions, only one of them
814 is actually going to be executed). Both branches must consume exactly the same resources.
815 - The S-ImplOne rules handles linear implications. The unrestricted and linear components of
816 the assumption are unioned with their respective context when solving the conclusion. Im-
817 portantly (see Section 6.3.2), the linear assumptions are added to the front of the list. The side
818 condition that the output context is a subset of the input context ensures that the implication
819 fully consumes its assumption and does not leak it to the ambient context.
820 - The S-ImplMany rules handles unrestricted implication. The conclusion uses its own linear
821 assumption, but none of the other linear constraints. This is because, as per C-Impl, unre-
822 stricted implications can only use an unrestricted context.

823
824 *6.3.2 An atomic-constraint solver.* So far, the atomic-constraint domain has been an abstract pa-
825 rameter. In this section, though, we offer a concrete domain which supports our examples.
826 For the sake of our examples, we need very little: linear constraints can remain abstract. It is thus
827 sufficient for the entailment relation (Figure 10a) to prove q if and only if it is already assumed—
828 while respecting linearity.
829 The corresponding atomic-constraint solver (Figure 10b) is more interesting. It is deterministic:
830 in all circumstances, only one of the three rules can apply. This means that the algorithm does not
831 guess, thus never needs to backtrack. Avoiding guesses is a key property of GHC's solver [Vytiniotis
832 et al. 2011, Section 6.4], one we must maintain if we are to be compatible with GHC.
833



(a) Entailment relation

(b) Atomic-constraint solver

Fig. 10. A stripped-down constraint domain

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

$$f = new\ 10\ (\lambda arr \rightarrow \mathbf{let}\ fr :: RW\ n \multimap ()\quad fr = free\ arr\quad () = fr\quad \mathbf{in}\ Ur\ ())$$

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

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

class C

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

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

$$bad :: C \multimap (Int, Int)$$
$$bad = (giveC\ useC, useC)$$

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].

883–931

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

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

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

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

Fig. 11. Core calculus (subset)

7 DESUGARING

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

7.1 The core calculus

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

- $\tau_1 \otimes \tau_2$ with sole constructor $(,) : \forall a\, b.a \rightarrow_1 b \rightarrow_1 a \otimes b$. 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 11 highlights the differences from the qualified system:

- Type schemes σ do not support qualified types.
- Existentially quantified types ($\exists \overline{a}.\tau \otimes Q$) are now represented as an (existentially quantified, linear) pair of values ($\exists \overline{a}.\tau_2 \otimes \tau_1$). Accordingly, **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[7].
- We have existential types. These can be realised in regular Haskell as a family of datatypes.

Using Lemma 6.4 together with Lemma 6.5 we know that if $\Gamma \mapsto e : \tau \rightsquigarrow C$ and $U; L \vdash_s C \rightsquigarrow \emptyset$, 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[8]. 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 12a. The $[\![_]\!]^{\mathbf{ev}}$ operation extends

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

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

(a) Evidence passing

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

(b) Desugaring (subset)

Fig. 12. Evidence passing and desugaring

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.

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

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

The cases correspond to the E-VAR, E-UNPACK[9], and E-SUB rules, respectively. Variables are stored with qualified types in the environment, so they get translated to functions that take the evidence as argument. Accordingly, the evidence is inserted by passing z as an argument. Handling **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:

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.*

[9]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.

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. So that in case there are multiple matching givens, the constraint solver selects the innermost one (in Section 6.3 we use an ordered list for this purpose).

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

8.2 Interaction with other features

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

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

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

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

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

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

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

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

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

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

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

8.3 Inferring 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[10] 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.

9 RELATED WORK

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

The solver judgement in OutsideIn takes the following form:

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

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. As discussed in Section 6, we do not perform type inference, only constraint inference. Therefore, our solver need not return a type assignment.

[10] 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.

1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099
1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127

- Both OutsideIn and our solver output a set of constraints, $Q_{residual}$ and L_o respectively. However, the meaning of these contexts is different. OutsideIn's *residual* constraints $Q_{residual}$ correspond to the part of C_{wanted} that could not be solved from the assumptions. These residuals are then quantified over in the generalisation step of the inference algorithm. We omit these residuals, which means that our algorithm cannot infer qualified types. Our *output* constraints L_o instead correspond to the part of the *linear* givens L_i that were not used in the solution for C_w.
- Finally, while OutsideIn has a single kind of conjunction, our constraint language requires two: $Q_1 \otimes Q_2$ and $Q_1 \& Q_2$. This shows up when generating constraints for **case** expressions in the rule G-Case rule. OutsideIn accumulates constraints across branches (taking the union of each branch), whereas we need to make sure that each branch of a **case**-expression consumes the same constraints.

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

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

Languages with capabilities. Both Mezzo [Pottier and Protzenko 2013] and ATS [Zhu and Xi 2005] served as inspiration for the design of linear constraints. Of the two, Mezzo is more specialised, being entirely built around its system of capabilities. ATS is the closest to our system because it appeals explicitly to linear logic, and because the capabilities (known as *stateful views*) are not tied to a particular use case. However, ATS does not have full inference of capabilities.

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

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

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

Logic programming. There are a lot of commonalities between GHC's constraint and logic programs. Traditional type classes can be seen as Horn clause programs, much like Prolog programs. GHC puts further 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.

10 CONCLUSION

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

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, 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.

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

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

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

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

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

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

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

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

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

1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225

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.

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

Fig. 13. Grammar of the core calculus

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

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

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

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

$$\textsc{L-Let}\ \frac{\Gamma_1 \vdash e_1 : \tau_1 \qquad \Gamma_2, x{:}_\pi \sigma \vdash e_2 : \tau}{\pi{\cdot}\Gamma_1 + \Gamma_2 \vdash \mathbf{let}_\pi\, x : \sigma = e_1\,\mathbf{in}\,e_2 : \tau}$$

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

Fig. 14. 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 13 and the type system by Figure 14.

A.2 Desugaring

The complete definition of the desugaring function from Section 7 can be found in Figure 15.

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

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

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

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

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

- either it is the conclusion of a C-Tensor rule, and the result is immediate.
- or it is the result of a C-Dom rule, in which case we have $C_1 = Q_1$, $C_2 = Q_2$, and the result follows from 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 \otimes 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 \otimes Q]\!]_{z_1}\ \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 \Rrightarrow \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 \Rrightarrow \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 \Rrightarrow \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. 15. 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 $C = \rho \cdot (Q_1 \mathrel{=\!\!\circ} C')$, 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 $(\pi \cdot \rho) \cdot Q' \vdash (\pi \cdot \rho) \cdot (Q_1 \mathrel{=\!\!\circ} C')$.
  In other words: $\pi \cdot Q \vdash \pi \cdot (\rho \cdot (Q \mathrel{=\!\!\circ} C))$ 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 $C = \rho \cdot (Q_1 \mathrel{=\!\!\circ} C')$, then $\pi \cdot C = (\pi \cdot \rho) \cdot (Q_1 \mathrel{=\!\!\circ} C')$. The result follows immediately by Lemma 6.1.

□

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

G-Var We have

- $\Gamma_1 = x{:}_1 \forall \overline{a}. Q \mathrel{=\!\!\circ} v$
- $\Gamma_1 + \omega \cdot \Gamma_2 \mapsto x : v[\overline{\tau}/\overline{a}] \leadsto Q[\overline{\tau}/\overline{a}]$
- $Q_g \vdash Q[\overline{\tau}/\overline{a}]$

  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

- $\Gamma \mapsto \lambda x.e : \tau_0 \to_\pi \tau \leadsto C$
- $Q_g \vdash C$
- $\Gamma, x{:}_\pi \tau_0 \mapsto e : \tau \leadsto C$

  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

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

  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

1422
- $Q_1; \Gamma_1 \vdash e_1 : \tau_1$

1423
- $Q_2; \Gamma_2, x{:}_\pi \tau_1 \vdash e_1 : \tau_1$

1424 From which follows that $Q_g; \pi \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}_\pi\ x = e_1\ \mathbf{in}\ e_2 : \tau$.

1425 G-LetSig We have

1426
- $\pi \cdot \Gamma_1 + \Gamma_2 \mapsto \mathbf{let}_\pi\ x : \forall \overline{a}. Q \Rightarrow\!\!\circ\ \tau_1 = e_1\ \mathbf{in}\ e_2 : \tau \rightsquigarrow C_2 \otimes \pi \cdot (Q \Rightarrow\!\!\circ\ C_1)$

1427
- $Q_g \vdash C_2 \otimes \pi \cdot (Q \Rightarrow\!\!\circ\ C_1)$

1428
- $\Gamma_1 \mapsto e_1 : \tau_1 \rightsquigarrow C_1$

1429
- $\Gamma_2, x{:}_\pi \forall \overline{a}. Q \Rightarrow\!\!\circ\ \tau_1 \mapsto e_2 : \tau \rightsquigarrow C_2$

1430 By Lemmas 6.1 and 6.3, there exist Q_1, Q_2 such that

1431
- $Q_2 \vdash C_2$

1432
- $Q_1 \otimes Q \vdash C$

1433
- $Q_g = \pi \cdot Q_1 \otimes Q_2$

1434 By induction hypothesis

1435
- $Q_1 \otimes Q; \Gamma_1 \vdash e_1 : \tau_1$

1436
- $Q_2; \Gamma_2, x{:}_\pi \forall \overline{a}. Q \Rightarrow\!\!\circ\ \tau_1 \vdash e_2 : \tau$

1437 Hence $Q_g; \pi \cdot \Gamma_1 + \Gamma_2 \vdash \mathbf{let}_\pi\ x : \forall \overline{a}. Q \Rightarrow\!\!\circ\ \tau_1 = e_1\ \mathbf{in}\ e_2 : \tau$

1438 G-App We have

1439
- $\Gamma_1 + \pi \cdot \Gamma_2 \mapsto e_1\ e_2 : \tau \rightsquigarrow C_1 \otimes \pi \cdot C_2$

1440
- $Q_g \vdash C_1 \otimes \pi \cdot C_2$

1441
- $\Gamma_1 \mapsto e_1 : \tau_2 \rightarrow_\pi \tau \rightsquigarrow C_1$

1442
- $\Gamma_2 \mapsto e_2 : \tau_2 \rightsquigarrow C_2$

1443 By Lemmas 6.1 and 6.3, there exist Q_1, Q_2 such that

1444
- $Q_1 \vdash C_1$

1445
- $Q_2 \vdash C_2$

1446
- $Q_g = Q_1 \otimes \pi \cdot Q_2$

1447 By induction hypothesis

1448
- $Q_1; \Gamma_1 \vdash e_1 : \tau_2 \rightarrow_\pi \tau$

1449
- $Q_2; \Gamma_2 \vdash e_2 : \tau_2$

1450 Hence $Q_g; \Gamma_1 + \pi \cdot \Gamma_2 \vdash e_1\ e_2 : \tau$.

1451 G-Pack We have

1452
- $\Gamma \mapsto \mathbf{pack}\ e : \exists \overline{a}. \tau \otimes Q \rightsquigarrow C \otimes Q[\overline{v}/\overline{a}]$

1453
- $Q_g \vdash C \otimes Q[\overline{v}/\overline{a}]$

1454
- $\Gamma \mapsto e : \tau[\overline{v}/\overline{a}] \rightsquigarrow C$

1455 By Lemma 6.1, there exist Q_1, Q_2 such that

1456
- $Q_1 \vdash C$

1457
- $Q_2 \vdash Q[\overline{v}/\overline{a}]$

1458
- $Q_g = Q_1 \otimes Q_2$

1459 By induction hypothesis

1460
- $Q_1; \Gamma \vdash e : \tau[\overline{v}/\overline{a}]$

1461 So we have $Q_1 \otimes Q[\overline{v}/\overline{a}]; \Gamma \vdash \mathbf{pack}\ e : \exists \overline{a}. \tau \otimes Q$. By rule E-Sub, we conclude $Q_g; \omega \cdot \Gamma \vdash \mathbf{pack}\ e :$
1462 $\exists \overline{a}. \tau \otimes Q$.

1463 G-Unpack We have

1464
- $\Gamma_1 + \Gamma_2 \mapsto \mathbf{unpack}\ x = e_1\ \mathbf{in}\ e_2 : \tau \rightsquigarrow C_1 \otimes 1 \cdot (Q' \Rightarrow\!\!\circ\ C_2)$

1465
- $Q_g \vdash C_1 \otimes 1 \cdot (Q' \Rightarrow\!\!\circ\ C_2)$

1466
- $\Gamma_1 \mapsto e_1 : \exists \overline{a}. \tau_1 \otimes Q' \rightsquigarrow C_1$

1467
- $\Gamma_2, x{:}_\pi \tau_1 \mapsto e_2 : \tau \rightsquigarrow C_2$

1468 By Lemma 6.1, there exist Q_1, Q_2 such that

1469
- $Q_1 \vdash C_1$

1470

1471 • $Q_2 \otimes Q' \vdash C_2$
1472 • $Q_g = Q_1 \otimes Q_2$
1473 By induction hypothesis
1474 • $Q_1; \Gamma_1 \vdash e_1 : \exists \overline{a}.\tau_1 \otimes Q'$
1475 • $Q_2 \otimes Q; \Gamma_2 \vdash e_2 : \tau$
1476 Therefore $Q_g; \Gamma_1 + \Gamma_2 \vdash \textbf{unpack}\ x = e_1\ \textbf{in}\ e_2 : \tau$.
1477 G-Case We have
1478 • $\pi \cdot \Gamma + \Delta \mapsto \textbf{case}_\pi\ e\ \textbf{of}\ \{\overline{K_i\ \overline{x_i} \longrightarrow e_i}\} : \tau \rightsquigarrow \pi \cdot C \otimes \& C_i$
1479 • $Q_g \vdash \pi \cdot C \otimes \& C_i$
1480 • $\Gamma \mapsto e : T\ \overline{\sigma} \rightsquigarrow C$
1481 • For each i, $\Delta, \overline{x_i{:}_{(\pi \cdot \pi_i)} v_i[\overline{\sigma}/\overline{a}]} \mapsto e_i : \tau \rightsquigarrow C_i$
1482 By repeated uses of Lemma 6.1 as well as Lemma 6.3, there exist Q, Q' such that
1483 • $Q \vdash C$
1484 • For each i, $Q' \vdash C_i$
1485 • $Q_g = \pi \cdot Q \otimes Q'$
1486 By induction hypothesis
1487 • $Q; \Gamma \vdash e : T\ \overline{\sigma}$
1488 • For each i, $Q'; \Delta, \overline{x_i{:}_{(\pi \cdot \pi_i)} v_i[\overline{\sigma}/\overline{a}]} \vdash e_i : \tau$
1489 Therefore $Q_g; \pi \cdot \Gamma + \Delta \vdash \textbf{case}_\pi\ e\ \textbf{of}\ \{\overline{K_i\ \overline{x_i} \longrightarrow e_i}\} : \tau$.
1490
1491 □
1492
1493 Proof of Lemma 6.5. By induction on $U; L_i \vdash_{\mathsf{s}} C \rightsquigarrow L_o$
1494 S-Atom We have
1495 • $U; L_i \vdash_{\mathsf{s}} \pi \cdot q \rightsquigarrow L_o$
1496 • $U; L_i \vdash_{\mathsf{s}}^{\mathsf{atom}} \pi \cdot q \rightsquigarrow L_o$
1497 By Property 6.6 we have
1498 (1) $L_o \subseteq L_i$
1499 (2) $(U, L_i) \Vdash \pi \cdot q \otimes (\emptyset, L_o)$
1500 Then by C-Dom we have $(U, L_i) \vdash \pi \cdot q \otimes (\emptyset, L_o)$.
1501 S-Add We have
1502 • $U; L_i \vdash_{\mathsf{s}} C_1 \& C_2 \rightsquigarrow L_o$
1503 • $U; L_i \vdash_{\mathsf{s}} C_1 \rightsquigarrow L_o$
1504 • $U; L_i \vdash_{\mathsf{s}} C_2 \rightsquigarrow L_o$
1505 By induction hypothesis we have
1506 • $L_o \subseteq L_i$
1507 • $(U, L_i) \vdash C_1 \otimes (\emptyset, L_o)$
1508 • $(U, L_i) \vdash C_2 \otimes (\emptyset, L_o)$
1509 Then by C-With we have $(U, L_i) \vdash C_1 \& C_2 \otimes (\emptyset, L_o)$.
1510 S-Mult We have
1511 • $U; L_i \vdash_{\mathsf{s}} C_1 \otimes C_2 \rightsquigarrow L_o$
1512 • $U; L_i \vdash_{\mathsf{s}} C_1 \rightsquigarrow L_o'$
1513 • $U; L_o' \vdash_{\mathsf{s}} C_2 \rightsquigarrow L_o$
1514 By induction hypothesis we have
1515 • $L_o \subseteq L_o'$
1516 • $L_o' \subseteq L_i$
1517 • $(U, L_i) \vdash C_1 \otimes (\emptyset, L_o')$
1518 • $(U, L_o') \vdash C_2 \otimes (\emptyset, L_o)$
1519

1520 Then by transitivity of $\subseteq$ we have $L_o \subseteq L_i$, and by C-Tensor we have $(U, L_i) \otimes (U, L'_o) \vdash$
1521 $C_1 \otimes C_2 \otimes (\emptyset, L'_o) \otimes (\emptyset, L_o)$ by Lemma 6.1 we have $(U, L_i) \vdash C_1 \otimes C_2 \otimes (\emptyset, L_o)$.
1522 S-ImplOne We have
1523 - $U; L_i \vdash_s 1 \cdot ((U_0, L_0) \multimap C) \rightsquigarrow L_o$
1524 - $U \cup U_0; L_i \uplus L_0 \vdash_s C \rightsquigarrow L_o$
1525 - $L_o \subseteq L_i$
1526 By induction hypothesis we have
1527 - $(U \cup U_0, L_i \uplus L_0) \vdash C \otimes (\emptyset, L_o)$
1528 - $L_o \subseteq L_i \uplus L_0$
1529 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
1530 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
1531 C-Tensor we conclude that $(U, L_i) \vdash 1 \cdot ((U_0, L_0) \multimap C) \otimes (\emptyset, L_o)$
1532 S-ImplMany We have
1533 - $U; L_i \vdash_s \omega \cdot ((U_0, L_0) \multimap C) \rightsquigarrow L_i$
1534 - $U \cup U_0; L_0 \vdash_s C \rightsquigarrow \emptyset$
1535 By induction hypothesis we have
1536 - $(U \cup U_0, L_0) \vdash C \otimes (\emptyset, \emptyset)$
1537 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
1538 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.
1539 □
1540
1541 Lemma B.2 (Weakening of wanteds). *If* $Q \vdash C$, *then* $\omega \cdot Q' \otimes Q \vdash C$
1542 Proof. This is proved by a straightforward induction on the derivation of $Q \vdash C$, using the
1543 corresponding property on the simple-constraint entailment relation from Definition 5.3, for the
1544 C-Dom case. □
1545
1546 Lemma B.3. *The following equality holds:* $\pi \cdot (\rho \cdot C) = (\pi \cdot \rho) \cdot C$.
1547 Proof. This is proved by a straightforward induction on the structure of C, using Lemma B.1
1548 for the case $C = Q$. □
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568