Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Verificación estática de contratos sobre tipos de sesión en Haskell

Tesis de Licenciatura en Ciencias de la Computación

Enzo Samuel Cioppettini

Director: Hernán Melgratti

Buenos Aires, 2022

ABSTRACT

El lenguaje de programación Haskell cuenta con diversas implementaciones de tipos de sesiones binarias. En este trabajo estudiamos la viabilidad de integrarlas con LiquidHaskell, una herramienta de verificación estática que extiende el lenguaje con tipos refinados. Si bien la estructura recursiva de las sesiones se puede codificar fácilmente mediante tipos paramétricos, garantizar tanto la dualidad como el uso linear de los canales requiere de extensiones del lenguaje de mayor complejidad. Nosotros partimos de implementaciones existentes, y que utilizan distintas extensiones y mecanismos. A partir de estas, en la medida en la que son compatibles con LiquidHaskell, exploramos la clase de contratos sobre la comunicación que se pueden escribir y verificar. Para esto utilizamos dos técnicas: una simple de integrar, pero que solo permite escribir contratos sobre cada mensaje, es decir, sin tener en cuenta los valores anteriormente intercambiados; y que permite expresar propiedades más ricas, incluyendo dependencias con los mensajes previos, pero cuya integración es más compleja.

Palabras claves: Tipos de sesión, tipos refinados, Haskell, sistemas de efectos, verificación estática, concurrencia.

Índice general

1. INTRODUCCIÓN

Los tipos de sesión se han consolidado en los últimos años como uno de los formalismos más exitosos para el análisis y verificación estática de protocolos de comunicación en sistemas compuestos por procesos que interactúan mediante el envío de mensajes [GR17]. Estos son una clase particular de tipos comportamentales que, en su formulación original [Hon93], permiten enforzar protocolos de comunicación entre dos participantes mediante el sistema de tipos; garantizando el uso complementario de los canales. Trabajos posteriores expandieron la formulación para protocolos que involucran más de dos participantes [HYC16].

Por otro lado, la metodología de Diseño por Contratos [Mey92], desarrollada originalmente para lenguajes de programación orientados a objetos; propone extender los tipos con predicados lógicos de alto nivel. Estos toman forma de precondiciones, poscondiciones e invariantes. Pueden usarse como documentación sobre la interfaz, y también como herramienta para garantizar la correctitud de la implementación. Para esto último existen diversas técnicas o metodologías. Por ejemplo, es posible generar validaciones en tiempo de ejecución (*assertions*) a partir de los predicados: [KHB99]. Por otro lado, para evitar el costo en tiempo de ejecución, también pueden usarse para generar tests: [HT10; Aic03], utilizando trazas arbitrarias que cumplan con las propiedades descritas, y validando que se mantengan los invariantes y poscondiciones. Alternativamente, es posible aplicar técnicas de análisis simbólico del código para probar estáticamente que el programa cumple con lo que promete: [Pey08; GF07].

Volviendo a los tipos de sesión, si bien estos expresan un contrato sobre el uso del canal de comunicación en cuestión, este solo predica sobre la estructura de esta, por lo cual lo único que sabemos sobre los mensajes es el tipo de cada uno. Si bien a partir de la estructura podemos derivar propiedades útiles, como la ausencia de deadlocks, es deseable en ciertos casos poder expresar propiedades más ricas sobre los mensajes enviados, potencialmente teniendo en cuenta los mensajes intercambiados anteriormente. Es por esto último que en múltiples instancias se ha estudiado la integración de los tipos de sesión con técnicas de diseño por contratos. Por ejemplo, en [MP17] se mostró que pueden aplicarse contratos chaperones a sesiones binarias, que monitorean la sesión en tiempo de ejecución para detectar (y reportar) cuando un módulo viola el contrato.

Por otro lado, también se ha estudiado la integración mediante técnicas de chequeo estático. Por ejemplo, se han extendido los tipos globales con aserciones para especificar restricciones sobre los valores comunicados en una sesión multipartes: [Boc+10; TY18]. Estos enfoques se basan en una metodología de arriba hacia abajo en la que toda la interacción entre múltiples partes se diseña a la vez y luego se proyecta sobre los participantes individuales de la sesión. En [Boc+10] se usa una lógica de decidible mientras que en [TY18] se usan tipos de sesión dependientes para verificar contratos. El chequeo estático de contratos para sesiones binarias en particular, fue estudiado por ejemplo en: [GG13]. En este se propone un enfoque que combina tipos dependientes LIQUID con los tipos sesión binarios y se presenta un algoritmo de inferencia de tipos para el sistema propuesto. Nuestra propuesta, tiene objetivos similares: enriquecer tipos de sesión binaria con tipos LIQUID, que permitan expresar restricciones de la interacción en función de los datos transmitidos. Sin embargo, nos proponemos abordar el estudio de la factibili-

dad de integrar implementaciones existentes de tipos sesión binaria y tipos LIQUID. En particular, consideraremos el caso del lenguaje de programación Haskell, que posee varias implementaciones de tipos sesión y además una extensión de tipos LIQUID denominada LiquidHaskell [Vaz+14]. Más concretamente, nuestras contribuciones son:

- Un análisis de la compatibilidad de LiquidHaskell con los mecanismos utilizados para implementar tipos de sesión en: [OY16; IYA11; KD21].
- Una demostración del tipo de contratos sobre una sesión que se pueden verificar utilizando LiquidHaskell para implementaciones que utilizan mecanismos compatibles, contemplando tanto contratos sensitivos como insensitivos a la historia.

A continuación damos una descripción de alto nivel de la estructura de este documento. En el capítulo 2 damos una breve introducción, desde el punto de vista de un usuario, a las funcionalidades de LiquidHaskell que utilizaremos para escribir los contratos. En el capítulo 3 damos una breve introducción a los tipos de sesión desde el punto teórico, y una noción a los problemas que podemos tener al implementarlos en Haskell. En capítulo 4 explicamos de forma general los mecanismos que provee el lenguaje que permiten tratar estos problemas y que son utilizados por las implementaciones que analizamos en el capítulo 5. Con respecto a este último, se encuentra dividido en cuatro secciones. En la primera mostramos el tipo más simple de contratos (sin dependencias) que nos gustaría poder escribir utilizando LiquidHaskell sobre una implementación de tipos de sesión. Cada una de las tres secciones siguientes se encuentra dedicada a una implementación existente de tipos de sesión: [OY16; IYA11; KD21], respectivamente. En primer lugar; analizamos los mecanismos, extensiones y técnicas que utiliza cada una, para luego analizar la viabilidad de combinar estos con LiquidHaskell para escribir los contratos de los que hablamos en la primera sección de este capítulo. Finalmente, en el capítulo 6 partimos de una de las implementaciones analizadas en el capítulo anterior, y analizamos la viabilidad de escribir contratos más ricos, enfocándonos en la posibilidad de escribir contratos que dependan de la historia de mensajes ya enviados.

2. LIQUIDHASKELL

LiquidHaskell es una herramienta que permite extender el sistema de tipos de Haskell con *Logically Qualified Data Types* o LIQUID types [RKJ08]. Estos permiten agregar a un tipo predicados que refinan (o reducen) sus valores posibles, luego mediante un sistema de inferencia de tipos dependientes se generan teoremas que se verifican mediante un *SMT solver*.

2.1. Refinamientos básicos

Los refinamientos se pueden especificar en el código haciendo uso de comentarios especiales. Por ejemplo, la siguiente expresión tiene tipo entero en lo que respecta a los tipos de Haskell:

```
someEven :: Int
someEven = 3
```

Si nos guiamos por su nombre, podemos asumir que retorna un número par, pero no tenemos forma de expresar esta propiedad en el tipo de la expresión. Utilizando LiquidHaskell, podemos refinar el tipo de la expresión para que el código anterior nos de un error en tiempo de compilación, agregando una anotación como la siguiente:

```
{-@ someEven :: {i:Int | i mod 2 = 0} @-}
```

En este caso, como 3 no es un número par, recibimos el siguiente error:

```
[!ht]
Liquid Type Mismatch
.
The inferred type
  VV : {v : GHC.Types.Int | v == (3 : int)}
.
is not a subtype of the required type
  VV : {VV : GHC.Types.Int | VV mod 2 == 0}
```

Que, a grandes rasgos, dice que en base a la implementación de *esPar*, LiquidHaskell infiere que el valor de retorno es 3; y que no vale que `3 mod 2` sea igual a 0. Si modificásemos el código de la expresión para que retornara un número par, e.g. 2, entonces no tendríamos ningún error.

También podemos agregar refinamientos sobre los tipos de entrada de una función, por ejemplo, `evenAddition` computa la suma pero solo de números pares, además de garantizar que el resultado también es un número par:

```
{-@ type Even = {i:Int | i mod 2 = 0} @-}
{-@ evenAddition :: Even -> Even -> Even @-}
evenAddition :: Int -> Int -> Int
evenAddition = +
```

Lo cual significa que obtendremos un error si intentamos llamarla con argumentos impares, por ejemplo:

```
{-@ f :: Even @-}
f = 2 `evenAddition` 3
```

Notar que si quitamos las precondiciones, es decir, cambiamos el tipo de la función por: `evenAddition :: Int -> Int -> Even`, ya no es posible probar que el resultado es par, por lo cual tendríamos un error de compilación. Si quisiéramos que no sea responsabilidad del usuario de la función garantizar que los argumentos son pares, podríamos cambiar la firma para aceptar cualquier par de números, y utilizar una implementación como la siguiente [1]:

```
{-@ evenAddition :: Int -> Int -> Maybe Even @-}
evenAddition x y =
     if x `mod` 2 == 0 && y `mod` 2 == 0
        then Just (x + y)
        else Nothing
```

En este caso, LiquidHaskell puede probar que si hay un resultado, este es par. También podríamos usar una poscondición más fuerte, estableciendo una dependencia entre el resultado y los parámetros. El siguiente refinamiento dice que se retorna un valor si y solo si los dos parámetros de entrada son pares, que también es demostrable a partir de la implementación anterior. Notar que en este caso estamos estableciendo una dependencia entre el resultado:**r** y los argumentos: **x** e **y**.

```
{-@ evenAddition ::
      x:Int ->
      y:Int ->
      {r:Maybe Even | (x mod 2 = 0 && y mod 2 = 0) <=> isJust r} @-}
```

2.2. Measures

En el caso anterior, *isJust* es lo que se define en LiquidHaskell como measure, que es un mecanismo que nos permite definir propiedades auxiliares sobre los valores. Son auxiliares porque son únicamente parte de la especificación, es decir, no existen en tiempo de ejecución. Para evaluarlas, se traducen como una función no interpretada cuando se forman las fórmulas que se pasan al SMT solver. Esto último significa que cuando se evalúa si la fórmula lógica resultante es satisfacible, la única propiedad que se utiliza es el nombre de la función y su aridad. Desde nuestro punto de vista, lo que podemos declarar son los valores de la measure para cada valor de entrada. En el caso de *isJust*, esta ya viene predefinida (notar que en este caso no es código Haskell, sino sintaxis específica de LiquidHaskell):

```
{-@ measure isJust :: Maybe a -> Bool
  isJust (Just x)  = true
  isJust (Nothing) = false @-}
```

[1] También podríamos probar que el resultado es par si los dos parámetros son impares, pero ignoramos ese caso para mantener el ejemplo compacto, ya que no es relevante para lo que nos interesa mostrar.

Notar que, si bien la *measure* no existe en tiempo de ejecución, el hecho de definirla en función de los constructores del tipo le permite a LiquidHaskell inferir su valor a partir de los valores que tengamos en nuestra implementación. Por otro lado, también es posible definir una *measure* como una función normal de Haskell, por ejemplo, el caso anterior podríamos escribirlo como:

```
{-@ measure isJust @-}
isJust (Just a) = True
isJust Nothing = False
```

En este caso, estamos definiendo tanto una función normal de Haskell (que por lo tanto podemos usar en el código), como una función del mismo nombre que podemos usar en la lógica.

Además de poder escribir funciones por casos, también podemos definir *measures* recursivas, por ejemplo, LiquidHaskell tiene también definida la *measure* **len**, que "mide" la longitud de una lista.

```
instance measure len :: forall a. [a] -> GHC.Types.Int
  len []     = 0
  len (y:ys) = 1 + len ys
```

Notar que el nombre es diferente a la función de Haskell **length**, que retorna el tamaño de una lista, sin embargo, estas dos están relacionadas directamente mediante el refinamiento con el que está anotada **length**:

```
length  :: xs:[a] -> {v: GHC.Types.Int | v = len(xs)}
```

Por otro lado, si bien en general se puede definir una *measure* en función de los constructores, existen ciertos casos donde esto no es posible, como cuando trabajamos con cierto tipo de estado o efectos secundarios. Por ejemplo, supongamos que abrimos un archivo mediante la función `openFile`, que toma un path, un modo de apertura (lectura, escrita, etc...), y retorna un objeto de tipo `Handle`. Supongamos luego que queremos definir una función que solo se pueda llamar con un `Handle` que fue abierto con permisos de escritura (`WriteMode`). En principio, dado solo el `Handle` no tenemos forma de expresar el predicado, ya que el modo de apertura era solo parte de la firma de `openFile`, y ya perdimos esa información. Para poder expresar este tipo de predicados, lo que podemos hacer es definir una *measure* sin dar su definición, solo su tipo, por ejemplo:

```
{-@ measure hMode :: Handle -> IOMode @-}
```

Por supuesto que al no dar una definición de la measure en función de los constructores, no hay forma de que LiquidHaskell infiera su valor a partir del código de una función, por lo cual para usarla tenemos que definir su relación con los valores. Una forma de hacerlo es utilizando *assume*, que es una directiva que nos permite declarar que cierto predicado vale sobre un tipo, sin que LiquidHaskell intente probarlo. Por ejemplo, podríamos definir nuestro propio wrapper sobre *openFile* que asocie el `Handle` con el modo de escritura, de la siguiente manera:

```
{-@ assume openFile :: FilePath ->
     m:IOMode -> IO {h:Handle | hMode h = m} @-}
openFile = System.IO.openFile
```

Esto también podríamos expresarlo utilizando el mecanismo anteriormente demostrado, definiendo un tipo nuevo como `data MHandle = MHandle IOMode Handle`, que "recuerde" el modo con el que el archivo fue abierto. Sin embargo, en este caso tendríamos un valor en runtime que solo necesitamos en la lógica, y deberíamos definir todas las funciones en función de este nuevo tipo.

Una vez que definimos nuestra versión de *openFile*, podemos por ejemplo definir una función que requiere un `Handle` abierto en modo de escritura, haciendo uso de la *measure*, como se muestra a continuación:

```
{-@ hPutStr :: {h:Handle | hMode h = WriteMode} -> String -> IO () @-}
hPutStr = System.IO.hPutStr
```

2.3. Abstracción

De la misma forma que en Haskell podemos usar polimorfismo paramétrico para escribir funciones genéricas, también es posible definir predicados abstractos que pueden instanciarse dependiendo de las circunstancias, permitiendo reutilizar código. Si quisiésemos definir una lista (o cualquier estructura) donde todos los elementos cumplen un predicado p, pero sin especificar cual es el predicado, podríamos definir algo como:

```
{-@ data RList a <p :: a -> Bool> = Nil | Cons (h:a<p>) (t:RList<p> a<p>) @-}
data RList a = Nil | Cons a (RList a)
```

Donde `h:a <p>` es azúcar sintáctico para `{h:a | p a}`. Luego, podemos trabajar con instancias particulares del predicado, por ejemplo:

```
{-@ evens :: RList <{\a -> a mod 2 = 0}> Int @-}
evens :: RList Int
evens = Cons 2 (Cons 4 Nil)
```

Si intentáramos retornar una lista con algún valor impar, entonces recibiríamos un error, por ejemplo, si en lugar del ejemplo anterior construimos la siguiente instancia del tipo:

```
evens = Cons 2 (Cons 3 Nil)
```

Al compilar recibimos un error que dice que 3 no es par:

```
Liquid Type Mismatch
.
The inferred type
  VV : {v : GHC.Types.Int | v == (3 : int)}
.
is not a subtype of the required type
  VV : {VV : GHC.Types.Int | VV mod 2 == 0}
```

Por otro lado, también podemos hacer uso del refinamiento para poder probar propiedades que dependen de los elementos de la lista, por ejemplo, podemos computar la sumatoria de los elementos de la lista, y probar que es par:

```
{-@ predicate IsEven N = (N mod 2) = 0 @-}
{-@ addEvens :: RList <{\a -> IsEven a}> Int -> { i:Int | IsEven i } @-}
addEvens :: RList Int -> Int
addEvens (Cons x xs) = x + (addEvens xs)
addEvens Nil = 0
```

2.4. Predicados abstractos acotados

Mientras que los refinamientos abstractos permiten reutilizar estructuras o funciones en diferentes contextos, esto solo nos permite definir tipos para los cuales cualquier predicado es válido, lo cual significa que no podemos asumir ninguna propiedad sobre ellos en nuestra implementación. Esto último es particularmente limitante cuando se trabaja con funciones de alto orden. Como solución a esto, LiquidHaskell tiene la noción de Bounded refinement types [VBJ15], que permiten *acotar* la cuantificación universal a los predicados que cumplen ciertas propiedades, de manera análoga a la que las typeclasses permiten expresar polimorfismo adhoc en Haskell. Una consecuencia importante de esto es que pueden utilizarse para definir y probar la composición de funciones, ya que dado el predicado del resultado de una composición, se pueden acotan los predicados de las funciones de entrada a aquellos cuya composición lo garantice. Esto último también puede extenderse a otras formas de componer funciones, por ejemplo a la composición de mónadas mediante *bind*.

3. TIPOS DE SESIÓN

Los tipos comportamentales permiten verificar que una implementación respeta un orden particular de operaciones, lo cual es útil cuando se trabaja con funciones con estado o side effects. En el presente trabajo nos concentramos en un tipo particular de tipos comportamentales: los tipos de sesión. Su función es la de restringir el uso de un canal de comunicación compartido de manera concurrente entre dos o más procesos. La característica distintiva es que, además de hablar de un orden de operaciones sobre un canal que permite enviar y recibir valores de un tipo que va cambiando, también queremos que este se use de manera complementaria: es decir, cuando uno de los participantes envía algo de un tipo por el canal, la otra parte debe recibir algo de ese tipo también. En el resto de este capítulo, damos una introducción informal a los tipos de sesión, para una introducción detallada, puede consultarse [Vas12].

Cuando hablamos de operaciones en el párrafo anterior, principalmente nos referimos a enviar valores a través de un canal, que generalmente en la literatura se nota como $!T.U$; y a recibir valores a través de un canal, que generalmente se nota $?T.U$. En el primer caso, estamos hablando del tipo de un canal que puede usarse para enviar un valor de tipo T, y luego puede usarse de acuerdo a U. El segundo es análogo, es un canal donde se recibe un elemento de tipo T y luego se puede usar de acuerdo a U. Esto nos permite construir secuencias como $!int.(?bool.end)$, que se lee como el tipo de un canal por el que primero se envía un entero, luego se recibe un booleano (presumiblemente con alguna relación al entero que se envió, por ejemplo, si es par), y luego no se puede utilizar más. Por otro lado, si bien en el fondo son casos particulares de enviar y recibir mensajes, es bastante común hablar de las operaciones que permiten establecer cierto control de flujo en la comunicación. Generalmente estas se notan como $\oplus\{l_i : T_i\}_{i \in I}$ y $\&\{l_i : T_i\}_{i \in I}$, donde el primero es el tipo que permite seleccionar una rama l_i de un conjunto finito de etiquetas, y el segundo significa que se le ofrece al otro lado ese conjunto de etiquetas a elegir. En ambos casos, tras elegida la etiqueta, se prosigue a usar el canal de acuerdo a T_i. Más adelante, cuando analicemos las implementaciones, nos concentraremos en las primitivas para enviar y recibir valores, pero cuando hablemos de los contratos dependientes en la capítulo 6 haremos uso de las primitivas de selección para analizar contratos sobre sesiones de tipo recursivo, que requieren de control de flujo para eventualmente terminar.

Aunque lo anterior solo restringe la implementación de los canales a aquellos mecanismos de comunicación que mantienen orden y no pierden mensajes, el mecanismo en particular no hace ninguna diferencia para el análisis realizado en este trabajo, por lo cual nos centraremos en canales implementados sobre memoria compartida.

En el listado 1 damos una primera aproximación a una representación de tipos de sesión en Haskell, utilizando tipos paramétricos recursivos para describir un protocolo de comunicación. Omitimos las implementaciones de las primitivas, ya que en el capítulo 5 mostraremos distintas formas de implementarlas. A continuación damos una descripción de alto nivel de la interpretación de cada tipo. `Send` y `Recv` son ambos *wrappers* sobre un canal, que solo puede usarse para ,respectivamente, enviar o recibir un valor de tipo `a`. El tipo `s` es en ambos casos el tipo con el que continua la sesión, que en principio podría ser `Send`, `Recv`, `End`, o alguna otra primitiva como la de selección de ramas que en este caso omitimos. Esto último no está expresado en el tipo en sí mismo, pero sí en las primitivas:

`send` y `recv`, que requieren que `s` sea un tipo de sesión a través de la clase (constraint) `Session`. Hablando de las primitivas, `send` toma un valor a enviar del mismo tipo que el primer parámetro que el tipo de sesión (a), que envía sobre el canal, y luego retorna una instancia del tipo de sesión siguiente. `recv` es similar, solo que en este caso no toma un valor a enviar, si no que retorna un par con el valor recibido, también de tipo a, y una instancia del tipo de la continuación (s). En el caso de `close` y de `End`, no hay tipos paramétricos ya que se usan para terminar la sesión; potencialmente pueden utilizarse para sincronizar la terminación y asegurarse que todos los mensajes sean consumidos por los dos participantes. Volviendo a hacer referencia a la notación que dimos anteriormente, el tipo $!int.(?bool.end)$ lo escribiríamos en este caso como `Send Int (Recv Bool End)`.

Por otro lado, tenemos `f1` y `f2`, que son usuarias de los tipos de sesión que acabamos de describir; es decir, son dos agentes secuenciales que deben ejecutarse de manera concurrente (por ejemplo, mediante dos threads) y se comunican entre sí utilizando canales con tipos de sesión. `f1` envía dos valores de tipo entero (2 y 4), y recibe un número de punto flotante, el cual luego retorna. El tipo del dominio de `f2` es el tipo dual del dominio de `f1`, es decir: recibe dos enteros y envía un valor de punto flotante, que es la división de los valores que recibió. Un problema con esta aproximación es que si bien intuitivamente podemos ver que los dominios son duales, esta relación no está expresada en los tipos, ya que no es posible hacerlo sin recurrir a ciertas extensiones de GHC que permiten escribir funciones a nivel de tipos. Las implementaciones que analizamos en el capítulo 5 adoptan diferentes maneras de representar esta relación.

Notar en ambos casos el uso del binding o variable *c*, que se reutiliza en cada paso de la sesión, haciendo *shadowing* del valor con el tipo anterior. Esta práctica previene que pueda repetirse cualquier paso del protocolo, ya que no hay forma de referenciar al canal una vez que ya se usó. El problema es que esto está subordinado al uso correcto por parte del programador, ya que no es enforzado por el sistema de tipado, por lo cual es posible violar el protocolo por error. Por ejemplo, no hay nada en el sistema de tipos que prevenga la implementación de `f2` demostrada en listado 2, que en particular causa un deadlock (cosa que de otra forma las sesiones previenen) si se compone paralelamente con la implementación previa de `f1`.

Más aún, tampoco hay nada que descarte la siguiente implementación, que ni siquiera utiliza el canal:

```
f2'' :: Recv Int (Recv Int (Send Float End)) -> IO ()
f2'' _ = return ()
```

Estos problemas pueden resolverse mediante el uso de un sistema de tipos lineales, que significa que los valores deben usarse exactamente una vez. Las implementaciones que analizamos en el capítulo 5 resuelven esto a través de extensiones del lenguaje, ya sea implementando un sistema de tipos lineares mediante funciones de tipos, o recurriendo a la extensión *LinearTypes* de GHC 9. En el capítulo 4 damos una introducción a estas extensiones, cuya aplicación sobre las sesiones podremos ver cuando analicemos cada implementación en el capítulo 5.

```
data Send a s = -- ...
data Recv a s = -- ...
data End = -- ...

class Session s where
  - ...

instance Session (Recv a s) where
  -- ...
instance Session End where
  -- ...
instance Session (Send a s) where
  -- ...

send :: Session s => a -> Send a s -> IO s
recv :: Session s => Recv a s -> IO (a, s)
close :: End -> IO ()

f1 :: Send Int (Send Int (Recv Float End)) -> IO Float
f1 c = do c <- send 2 c; c <- send 4 c; (f, c) <- recv c; close c; return f

f2 :: Recv Int (Recv Int (Send Float End)) -> IO ()
f2 c = do (a, c) <- recv c
          (b, c) <- recv c
          c <- send (int2Float a / int2Float b) c
          close c
```

Listing 1: Pre-sesiones sin dualidad

```
f2' :: Recv Int (Recv Int (Send Float End)) -> IO ()
f2' c = do
    (a, c') <- recv c
    (b, c') <- recv c'
    -- el siguiente recv utiliza el canal con el 'estado'
    -- inicial de vuelta
    recv c
    c' <- send (int2Float a / int2Float b) c'
    close c'
```

Listing 2: Uso no linear de los tipos de sesión

4. EXTENSIONES DE GHC/HASKELL

4.1. Programación a nivel de tipos

Las versiones recientes de GHC poseen diversas herramientas que pueden usarse para computar tipos, lo cual es particularmente relevante en el contexto de la implementación de sesiones por dos razones, una es que nos permite expresar la noción de dualidad, que es una relación entre tipos. La otra es que pueden usarse para implementar sistemas de efectos, que pueden usarse para validar el uso linear de los canales. La idea de esta sección es dar una introducción a estas extensiones y conceptos clave.

4.1.1. Kinds

Cuando estamos programando a nivel de tipos, es decir, usando constructores de tipos para generar nuevos tipos, podemos pensar en los *kinds* como el análogo de los tipos cuando programamos a nivel de términos, es decir, son los "tipos de los tipos". El sistema de kinds de GHC se encuentra activo por defecto sin necesidad de habilitar extensiones (a partir de ghc 8), y tiene ciertos kinds incluidos. El más elemental es $*$ (o *Type* en GHC 9), que es el kind de los tipos que tienen términos [1]. Es el kind de, por ejemplo, **Bool** e **Int**. Más generalmente, es el kind de los tipos que podemos asignar a una expresión. Los constructores de tipos como `Maybe` o `[]` tienen kind $* \to *$, lo cual implica que sus aplicaciones, como `Maybe` o `[Int]`, tienen kind $*$. Por otro lado, también existe el **Constraint**, que es el kind de las typeclasses aplicadas, es decir, **Eq** tiene kind $* \to$ *Constraint* y **Eq Int** tiene kind **Constraint**.

4.1.2. DataKinds

DataKinds nos permite extender los kinds incluidos con GHC, promoviendo las declaraciones de tipos usuales mediante **data**. Por ejemplo, lo siguiente nos permite definir los naturales a nivel de tipos [2].

```
data Nat = Zero | Succ Nat
```

Es decir, si se encuentra habilitada la extensión, además de definir el tipo `Nat` con los naturales como valores, también se definen los **tipos** `Zero` y `Succ`, que tienen *kind* `Nat` y `Nat` $\to$ `Nat` respectivamente.

Algo que uno podría preguntarse es cuál es el propósito de definir estos tipos que no tienen valores, y una de las principales razones es que podemos usarlos para anotar funciones y estructuras de datos con propiedades extra. Por ejemplo, como se muestra en listado 3, podemos tener una noción en tiempo de compilación del tamaño de una lista, estableciendo en este caso poscondiciones en sus constructores. En este caso `SList` está compuesto internamente por una lista estándar del lenguaje (`[a]`), por lo cual su

[1] Podemos pensar que el tipo `Void` tiene 0 términos. Si bien no podemos construir una instancia, podemos tipar una expresión como `Void`, por ejemplo, la expresión: `f = f`. Esto último no podemos hacerlo con los constructores como `Maybe`.

[2] Esto es a modo de ejemplo, ya que GHC tiene incluida una definición de los naturales a nivel de tipos, por lo cual no hace falta que los definamos.

representación como estructura de datos es la misma, la diferencia es que además de tener el parámetro de tipo `a`, que es el tipo de los elementos de la lista, tiene un parámetro extra (size) de kind `Nat`. Este último no representa el tipo de un valor que exista en memoria en tiempo de ejecución (no es de kind $*$), por eso no aparece del lado derecho de la definición; su único propósito es el de tener una anotación del tamaño de la lista en tiempo de compilación. Las funciones `cons` y `empty` actúan como constructores, cuya implementación simplemente delega a los constructores de la lista (`[]` y `(:)`), la parte importante es que instancian el parámetro `size` como corresponde en cada caso. La lista vacía tiene tamaño 0, por lo cual `empty` retorna: `SList a Zero`. Por otro lado, agregar un elemento a la lista incrementa el tamaño en 1, por eso el uso de `Succ` en el tipo de `cons`.

```
newtype SList a (size :: Nat) = SList [a]

empty :: SList a Zero
empty = SList []

cons :: a -> SList a s -> SList a (Succ s)
cons x (SList xs) = SList (x:xs)
```

Listing 3: Lista etiquetada con su tamaño como un tipo fantasma

4.1.3. TypeFamilies

Poder definir tipos y kinds es particularmente útil en combinación con **TypeFamilies**, que nos permite definir el equivalente a las funciones cuando programamos en términos. Por ejemplo, uno podría definir una función de comparación $>$ de naturales a nivel de tipos, como se muestra en el listado 4. Podemos pensar que `Gt` es análoga a una función de tipo `Nat -> Nat -> Bool`, solo que al ser una type family los argumentos son tipos, de kind `Nat`, y el resultado también es un tipo, de kind `Bool`.

4.1.4. Type Equalities

Además de establecer anotaciones sobre las poscondiciones de funciones mediante constructores y type families, también podemos establecer precondiciones, mediante el uso del kind **Constraint**. Por ejemplo, si escribiésemos:

```
f :: (Eq a) => a -> a -> Bool
f = (==)
```

Lo que estamos pidiendo en este caso como precondición sobre `a` es que implemente una interfaz, la de `Eq`. La extensión de **TypeFamilies** también nos permite definir un nuevo

```
type family Gt (a :: Nat) (b :: Nat) :: Bool where
    Gt 'Zero b = 'False
    Gt ('Succ a) 'Zero = 'True
    Gt ('Succ a) ('Succ b) = Gt a b
```

Listing 4: Implementación de $>$ mediante *TypeFamilies*

```haskell
type Two = (Succ (Succ Zero))

getThird :: (Gt s Two ~ True) => SList a s -> a
getThird (SList xs) = xs !! 3

test :: Int
test = getThird $ cons 1 (cons 2 (cons 3 empty))

-- esto no compila
-- testInvalid = getThird (cons 2 (cons 3 empty))
```

Listing 5: Acceso seguro a una lista etiquetada mediante *TypeFamilies*

tipo de constraint: la igualdad de tipos (∼). Por ejemplo, la expresión `a ~ Zero` tiene kind `Constraint`, y requiere que el tipo paramétrico `a` sea exactamente `Zero`. Podemos usar esto por ejemplo para definir una función que requiere una lista con más de dos elementos, permitiendo acceder mediante índices de manera segura, haciendo uso de la definición anterior de **Gt**, como se muestra en el listado 5.

4.1.5. FunctionalDependencies

Si bien las type families son el mecanismo más reciente para describir relaciones de tipos en Haskell, la extensión **FunctionalDependencies** también permite definir relaciones entre tipos, mediante typeclasses con más de un parámetro. La extensión nos permite decir que algunos parámetros están determinados por otros. Un análogo al ejemplo definido con *TypeFamilies*, se puede escribir como se muestra en el listado 6.

Como se puede notar, en este caso también necesitamos de otras extensiones. **FlexibleContexts** nos permite utilizar los tipos en el constraint, ya que ordinariamente solo se pueden utilizar variables de tipo, y **FlexibleInstances** cumple un rol similar para las instancias. **UndecidableInstances** es necesaria para expresar el caso recursivo, puesto que si bien en este caso termina gracias a la dependencia funcional, las reglas que utiliza GHC por defecto para asegurar la terminación no permiten demostrarlo. Una definición más formal de las reglas de terminación se puede encontrar en [Sul+06].

4.2. LinearHaskell

A continuación, damos una breve introducción a los tipos lineares implementados en GHC 9, para una descripción más completa puede consultarse [Ber+17]. Cuando se habilita la extensión: **LinearHaskell**, es posible anotar los argumentos de las funciones de acuerdo a si se utilizan de manera linear o no, por ejemplo, una función de tipo `cons` tiene que garantizar que el primer argumento se utiliza de manera linear (es decir, exactamente una vez) en el cuerpo de la función. En listado 7 se muestra una posible implementación para este tipo, y una que no lo es.

Vale la pena notar que la restricción de linearidad solo importa si el resultado de una función se utiliza de manera lineal, es decir, que es posible llamar a funciones lineares desde funciones no lineares, como se puede ver en la implementación de `g` en listado 8.

```haskell
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE UndecidableInstances #-}
{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE KindSignatures #-}

class Gt (a :: Nat) (b :: Nat) (c :: Bool) | a b -> c

instance Gt 'Zero n' 'False
instance Gt ('Succ n) 'Zero 'True
instance Gt n n' b => Gt ('Succ n) ('Succ n') b

type Two = ('Succ ('Succ 'Zero))
getThird :: (Gt s Two 'True) => SList a s -> a
getThird (SList xs) = xs !! 3

test :: Int
test = getThird $ cons 1 (cons 2 (cons 3 empty))

-- esto no compila
-- testInvalid :: Int
-- testInvalid = getThird (cons 2 (cons 3 empty))
```

Listing 6: Acceso seguro a una lista, enforzado mediante *FunctionalDependencies*

```haskell
f :: Int %1 -> Int -> Int
f a b = a + b + b
-- lo siguiente no compila, por ejemplo
-- f a b = a + a + b
```

Listing 7: Función lineal en el primer argumento y contraejemplo.

```haskell
g :: Int -> Int -> Int
-- Nota: la versión point free no tipa
g a b = f a b
```

Listing 8: Función no lineal que llama a una lineal en un argumento sin problemas.

Por otro lado, debido a que la biblioteca estándar de Haskell (base) no tiene funciones anotadas con tipos lineales, existe un preludio alternativo (o complementario) llamado *linear-base*, que define variantes lineales de las funcionalidades que se encuentran en base. Entre otras: una versión lineal de `IO`; enteros y aritmética de tipos lineales; y definiciones alternativas de `Functor` `Monad`. Por otro lado, también podemos encontrar en *linear-base* algunas abstracciones útiles para trabajar con tipos lineales, que no tienen un equivalente en base por no ser necesarias. Por ejemplo, hay ciertas clases que nos permiten realizar operaciones no lineales en un contexto lineal. La clase `Consumable` permite ignorar un valor

```
1  send :: Session s => a %1 -> Send a s %1 -> Linear.IO s
2  recv :: Session s => Recv a s %1 -> Linear.IO (a, s)
3  close :: End %1 -> Linear.IO ()
4
5  f2 :: Recv Int (Recv Int (Send Float End)) %1 -> Linear.IO ()
6  f2 c = do
7      (a, c') <- recv c
8      (b, c') <- recv c'
9      -- el siguiente recv utiliza el canal con el 'estado'
10     -- inicial de vuelta, pero gracias a LinearTypes no compila
11     (a, c) <- recv c
12     c' <- send (result (move a) (move b)) c'
13     close c'
14     where
15         result :: Ur Int %1 -> Ur Int %1 -> Float
16         result (Ur a) (Ur b) = int2Float a / int2Float b
```

Listing 9: Modificación del ejemplo del listado 2 para usar *LinearTypes*

de tipo lineal, mientra que la clase `Dupable` nos permite hacer copias de un valor. Estas últimas son principalmente útiles cuando necesitamos interactuar con funciones puras que no están en un contexto lineal, o en el caso de `Consumable`, cuando queremos poder ignorar algún valor, como por ejemplo el resultado de una función que retorne `()`, cuyo valor no aporta información en sí mismo.

Utilizando estas herramientas podemos solucionar el problema ilustrado en el listado 2, cambiando los tipos como mostramos en listado 9, ya que en este caso (correctamente) no compila debido a la línea 11. Como la división está definida para tipos no lineales (`g`), en este caso utilizamos `move :: a %1 -> Ur a` de la clase *Movable*, que nos permite convertir un tipo copiable (`Dupable`) con multiplicidad lineal en uno sin restricciones. Vale la pena aclarar que no implementamos esta clase para las sesiones, ya que no queremos que puedan utilizarse de manera irrestricta.

5. CONTRATOS E IMPLEMENTACIÓN DE SESIONES

5.1. Contratos

En el capítulo 3 vimos cómo los tipos de sesión describen un protocolo de comunicación y permiten validar que es respetado por ambas partes. Sin embargo, estas garantías están limitadas al orden de los mensajes, y no dicen nada sobre el contenido de estos. Continuando con el ejemplo de `f2` del listado 1, podríamos querer restringir el segundo parámetro a los números no negativos, puesto que de otras forma la división no está definida, y expresar esto como una propiedad de la sesión que `f1` tiene que cumplir para utilizarla. En principio, y de manera análoga a la propuesta en [GG13], uno podría querer expresar lo anterior mediante un tipo como el siguiente:

```
f1 :: Send Int (Send {i:Int | i != 0} (Recv Int End))
```

Alternativamente, lo que podemos hacer es utilizar un nuevo tipo y poner el refinamiento en el constructor, por ejemplo:

```
data NonZero = NZ Int
{-@ NZ :: {i:Int | i != 0 } -> NonZero @-}
```

Y luego, al ser un tipo de Haskell podemos simplemente cambiar el tipo de f1 sin necesidad de agregar ningún refinamiento, ya que la propiedad se valida cuando se utiliza el constructor, en este caso:

```
f1 :: Send Int (Send NonZero (Recv Int End))
```

En la siguiente sección, en primer lugar analizamos la viabilidad de aplicar estas dos técnicas para escribir contratos sobre las sesiones, refiriéndonos a la primera como agregar *refinamientos inline*, y a la segunda como utilizar *tipos etiquetados*.

5.2. Effect-sessions

Orchard y Yoshida proponen en [OY16] una implementación de tipos de sesión con linearidad mediante una mónada parametrizada con efectos. Los sistemas de efectos extienden las reglas de tipado estándar con anotaciones de carácter descriptivo, cuyo propósito es el análisis estático. Una forma de implementarlos en Haskell, y que es la propuesta en esta implementación, es mediante graded monads, que son una generalización de las mónadas. La principal diferencia es que se utiliza un parámetro de tipo adicional para describir los efectos de un cómputo, es decir, si normalmente interpretamos un valor de tipo $m\,a$ (donde m es una mónada) como un cómputo que retorna un valor de tipo a y tiene un efecto de tipo m, ahora interpretamos $m\,f\,a$ como un valor que representa un cómputo que retorna algo de tipo a, y tiene el efecto mf. En cuanto a la representación de los efectos (f), es importante resaltar que necesitamos combinarlos cuando componemos, para poder expresar el efecto del resultado. Como la composición (*bind*) es asociativa, necesitamos que la composición de dos efectos también lo sea, de lo contrario podríamos tener dos

expresiones cuya ejecución sea la misma, pero que presenten efectos diferentes. Además, para poder expresar operaciones puras (como *return*), también necesitamos un tipo que represente la identidad (de la composición de efectos), lo cual da lugar a la estructura de un monoide. Son particularmente idóneas para describir contratos o protocolos enforzados por el algoritmo de tipado.

5.2.1. Typeclass Effect

Para definir las *graded monads*, se utiliza la clase mostrada en listado 10. A continuación, explicamos sus componentes. El *kind* de **m** nos dice que el tipo que instancia la *typeclass* debe ser un constructor de tipos de dos parámetros, donde el primero tiene que ser el efecto, y que tiene *kind* k. `Unit`, `Plus` e `Inv` son *type families* asociadas; pueden interpretarse como funciones de tipos donde los valores de entrada son las instancias de `Effect`. La parte importante es que permiten que cada instancia defina su propia forma de representar y combinar los efectos. Por ejemplo, una instancia podría tener el efecto de un contador, en cuyo caso `Plus` podría ser la suma de enteros (a nivel de tipos), mientras que otra instancia podría tener un conjunto como efecto, y definir `Plus` como la unión. En el caso de las sesiones, tenemos una sola instancia, que mostramos en listado 11.

`Unit` y `Plus` son las operaciones que definen la familia de tipos que representan los efectos, que como ya se mencionó antes, tiene la estructura de un monoide, siendo `Unit` la identidad y `Plus` el operador binario asociativo. Notar que ambas están definidas completamente en función de k, que es el *kind* del primer parámetro, que es el "tipo" de los efectos.

`Inv` establece un invariante como un predicado, que por defecto es siempre verdadero (el *constraint* vacío). Es ligeramente diferente de las dos anteriores, ya que establece una restricción en lugar de una anotación. [1]

Una contra de utilizar una *typeclass* nueva, es que se pierden conveniencias como la capacidad de usar do notation, pero es posible de recuperar con la extensión **RebindableSyntax**, que permite que GHC utilice las definiciones de `>>=` y return que se encuentren en scope, en lugar de las del preludio, que son las de Monad.

5.2.2. Instancia de Effect: Process

Para la instancia particular de tipos de sesión, se utiliza la definición que mostramos en el listado 11. Omitimos la implementación, dado que principalmente nos interesan los efectos expresados en los tipos.

El tipo (**s**) con el que se encuentra parametrizada es un mapa finito de nombres de canales a tipos de sesión, representado como una lista de pares. El nombre del canal es un string con una etiqueta que identifica el "lado" de la sesión en el cual se usa. `Symbol` es el `kind` de las *strings* a nivel de tipos.

`UnionSeq` se comporta como una unión de conjuntos en el caso de que se compongan dos sesiones que utilicen canales diferentes. En el caso en el que los nombres de los canales (la clave del mapa) coinciden, se componen secuencialmente las sesiones (que son los valores del mapa). En el listado 13 se pueden ver algunos ejemplos del resultado de esta operación. Notar que es asociativa, ya que tanto la concatenación de listas (o la composición secuencial sesiones) como la unión de conjuntos lo son.

[1] Si se instancian `Unit` con (), `Plus` con () () e `Inv` con (); la estructura es equivalente a la de Monad.

```
{-# LANGUAGE KindSignatures,
             TypeFamilies,
             ConstraintKinds,
             PolyKinds,
             MultiParamTypeClasses
#-}

class Effect (m :: k -> * -> *) where
   type Unit m :: k
   type Plus m (f :: k) (g :: k) :: k

   type Inv m (f :: k) (g :: k) :: Constraint
   type Inv m f g = ()

   return :: a -> m (Unit m) a
   (>>=) :: (Inv m f g) => m f a -> (a -> m g b) -> m (Plus m f g) b
   (>>) :: (Inv m f g) => m f a -> m g b -> m (Plus m f g) b
```

Listing 10: Implementación typeclass Effect

```
data Process (s :: [Map Name Session]) a = Process { getProcess :: IO a }

instance Effect Process where
   type Plus Process s t = UnionSeq s t
   type Unit Process     = '[]
   type Inv Process s t  = Balanced s t

   return :: a -> Process (Unit Process) a

    -- TODO: no sé si este salto de línea compila
   (>>=) :: (Inv Process s t) => Process s a -> (a -> Process t b)
        -> Process (Plus Process s t) b
```

Listing 11: Instancia de Effect

```haskell
{-| Channel endpoint names -}
data Name = Ch Symbol | Op Symbol
{-| Named channels, encapsulating Concurrent Haskell channels -}
data Chan (n :: Name) = forall a . MkChan (C.Chan a)
```

Listing 12: Definición de canales

```haskell
type A = ('Ch "c" ':-> (Int ':? 'End)) ': '[]
type B = ('Ch "c" ':-> (Bool ':! 'End)) ': '[]
-- Resultado de: UnionSeq A B
type AuB = ('Ch "c" ':-> (Int ':? (Bool ':! 'End))) ': '[]
```

Listing 13: Ejemplo unión secuencial de sesiones

En Haskell, los tipos anteriores se pueden representar de la siguiente manera (reducida a solo `send` y `recv` por simplicidad). La extensión `DataKinds` permite utilizar los constructores como tipos (de kind `Session`), y en este caso la definición de `Session` es exclusiva para esto, puesto que no se utilizan a nivel de términos.

```haskell
data Session =  forall a . a :! Session  | forall a . a :? Session  | End
```

Listing 14: Representación del tipo sesión

A continuación mostramos los tipos de las primitivas. El tipo de `send` es el siguiente:

```haskell
send :: Chan c -> t -> Process '[c :-> t :! End] ()
```

Dice que toma un canal de nombre **c** y un elemento a enviar de tipo **t**; y retorna un proceso anotado con un ambiente que consiste de un único canal **c**, asociado con el tipo sesión $t\,!\,end$, cuya interpretación es que se envía un elemento de tipo **t** y luego termina.

El tipo de `recv` es análogo, solo que en este caso no toma un valor, sino que lo retorna, y la anotación del entorno dice que se recibe un valor de tipo **t** y luego termina la sesión:

```haskell
recv :: Chan c -> Process '[c :-> t :? End] t
```

La función `new` se usa para crear los canales, esta recibe una función que requiere un par de canales y los utiliza de manera dual (garantizado por `Duality`). Su tipo es el siguiente.

```haskell
new :: (Duality env c)
          =>  ((Chan (Ch c), Chan (Op c)) -> Process env t)
          ->  Process (env :\ (Op c) :\ (Ch c)) t
```

Cabe destacar que no retorna el canal, sino que se lo pasa a una función, esto garantiza que los canales se utilicen solo dentro del contexto donde está garantizado que se usan de manera dual. `Duality` es un predicado (representado mediante un `Constraint`) que dice que el canal c se usa de manera dual en env, en listado 15 podemos ver un ejemplo de un tipo que cumple con la propiedad para un canal >>=. El operador de `UnionSeq` es el

```
-- Env cumple Duality con "c" (no 'Ch "c" u 'Op "c")
type Env = ('Ch "c" ':-> (Int ':? 'End))
           ': ('Op "c" ':-> (Int ':! 'End)) ': '[]
```

Listing 15: Ejemplo Duality

que elimina las sesiones del canal en el mapa, esto simboliza que los canales se usan en su totalidad, y permite reusar el nombre en un nuevo canal si así se quisiera.

La composición paralela de procesos está implementada mediante `par`, y su tipo es el siguiente:

```
par :: (BalancedPar env env') =>  Process env () -> Process env' ()
     -> Process (DisjointUnion env env') ()
```

El constraint `BalancedPar`, significa que si los canales tienen nombres duales, entonces también tienen que tener tipos de sesión duales; el efecto del resultado simplemente une los dos entornos. La parte de "disjunta" es para diferenciarla de la unión utilizada en `Plus` (`UnionSeq`), que usa una función para combinar las sesiones cuando las claves coinciden. La implementación es la de una unión normal de conjunto, donde la unicidad es de el par clave-valor y no de las claves.

Cabe destacar que mientras analizábamos la biblioteca notamos que la implementación `BalancedPar` no verifica realmente que los entornos sean disjuntos, ya que si bien garantiza que los canales duales (con el mismo nombre) tengan tipos duales, permite que el mismo canal se use en ambos procesos, lo cual consideramos se debe a un error de implementación, ya que permite tipar sesiones que no usan los canales de forma lineal, como mostramos en el listado 16. [2]

```
d1 (c :: (Chan (Ch "c"))) = do
  send c ()
  send c ()
  recv c

d2 (c :: Chan (Op "c")) (d :: Chan (Ch "c")) = do
  recv c
  recv c
  send c ()
  recv d

d2d1 = new $ \(c, c') -> d1 c `par` d2 c' c
```

Listing 16: Composición paralela sin tipos disjuntos (bug)

[2] Esto fue comunicado a uno de los autores, aunque sin respuesta.

```
data NonZero = NZ Int deriving (Show)
{-@ NZ :: {i:Int | i > 0 } -> NonZero @-}

divServer (c :: Chan (Ch "c")) (d :: (Chan (Ch "d"))) = do
  x      <- recv c
  (NZ y) <- recv c
  send d (x `div` y)

divClient (c :: Chan (Op "c")) (d :: (Chan (Op "d"))) = do
  send c (2 :: Int)
  send c (NZ 0) -- unsound
  answer <- recv d
  putStrLn $ "result " ++ show answer
```

Listing 17: Ejemplo liquid unsound

5.2.3. Integración con LiquidHaskell

Desafortunadamente, el soporte para código que utiliza *type families* en LiquidHaskell es minimal y experimental, lo cual se puede observar en la suite de tests.

Además de eso, debido a que LiquidHaskell no soporta la sintaxis que se utiliza para escribir los tipos de los efectos, (e.g. listas a nivel de tipos), y teniendo en cuenta que los tipos enviados o recibidos se encuentran descritos únicamente ahí, no es posible escribir refinamientos *inline* sobre los tipos en cuestión. La prueba más simple que se realizó fue la de enviar elementos etiquetados para establecer contratos no dependientes de los valores anteriores de la sesión, como se muestra en listado 17. La razón por la cual es necesario utilizar dos canales, uno para enviar y uno para recibir, es que los canales en esta implementación se reusan a través de la sesión, lo cual significa que si hacemos un *send* seguido de un *recv*, podríamos leer el valor que acabamos de enviar antes de que el otro participante lo haga.

Sin embargo, LiquidHaskell ignora el hecho de que se está pasando cero (0) al constructor en la segunda línea de *divClient*, a pesar de que si se escribe la misma expresión por fuera del contexto de la sesión, la rechaza sin problemas.

A raíz de lo anterior, se construyó un ejemplo reducido que se reportó a los desarrolladores de LiquidHaskell, que demuestra que basta con utilizar las extensiones: *DataKinds*, *TypeFamilies*, *PolyKinds*, que forman un subconjunto de las utilizadas en esta implementación, para escribir código que se valida incorrectamente, es decir, un ejemplo de *unsoundness*. Por ejemplo, el código en listado 18 es *safe* aunque no debería.

Lo anterior hace bastante difícil el análisis, por lo cual se decidió no seguir trabajando sobre esta implementación, ya que no tener la confianza de que el código realmente se está verificando va en contra de la idea de usar verificación estática.

```haskell
{-# LANGUAGE DataKinds, TypeFamilies, PolyKinds #-}

newtype EMonad (r :: ()) a = EMonad { getInner :: IO a }

type family Plus (e :: k -> * -> *) (f :: k) (g :: k) :: k where
  Plus EMonad _ _ = '()

type family Unit (e :: k -> * -> *) :: k where
  Unit EMonad = '()

data NonZero = NZ Int deriving (Show)
{-@ NZ :: {i:Int | i > 0 } -> NonZero @-}

eBind :: EMonad s a -> (a -> EMonad t b) -> EMonad (Plus EMonad s t) b
eBind x k = EMonad ((>>=) (getInner x) (getInner . k))

-- divClient ::  Int -> EMonad '() () -- this makes it unsafe
divClient _ = send (NZ 0) `eBind` (\_ -> EMonad $ return ())

send :: t -> EMonad '() ()
send _ = EMonad $ return ()
```

Listing 18: Reproducción de un bug en LiquidHaskell, como se reportó en 1833

5.3. Full Sessions

Luego de descartar la implementación previamente analizada decidimos analizar *Full Sessions* [IYA11], que extiende a su vez trabajos anteriores [PT09].

En este caso, la implementación del sistema de efectos se basa en una mónada parametrizada con un par de tipos *ss* y *tt*, que describen un entorno o estado, antes y después de efectuar el cómputo, respectivamente:

```
newtype Session t ss tt a = Session { session :: ss -> IO (tt,a) }
```

Las precondiciones y poscondiciones se materializan mediante clases con dependencias funcionales, que se usan como *constraints* para describir los cambios de estado de cada operación.

Por ejemplo, la operación de **send** tiene el siguiente tipo:

```
send :: (Pickup ss n (Send v a), Update ss n a ss', IsEnded ss F)
     => Channel t n -> v -> Session t ss ss' ()
```

Donde **n** representa el índice del canal sobre el que se está operando, ya que la implementación soporta el uso de más de un canal. El primer componente del constraint (`Pickup`) requiere que en el canal **n** del entorno **ss** se encuentre una lista a nivel de tipos, que comienza con `Send v` y continúa con **a**. La segunda componente (`Update`) dice que en el entorno de salida **ss'**, el canal **n** tiene tipo **a**. La tercera componente valida que la sesión no haya terminado (`F` es false), es decir, forma parte de la precondición. `Pickup`, `IsEnded` y `Update` se implementan mediante dependencias funcionales, donde el último parámetro es determinado por los anteriores, lo cual significa que puede interpretarse como la salida de la función. El tipo de **recv** es similar, reemplazando `Send v` por `Recv v`:

```
recv :: (Pickup ss n (Recv v u), Update ss n u ss', IsEnded ss F)
     => Channel t n -> Session t ss ss' v
```

Los canales se crean mediante **new**, que asigna el índice del canal en base al tamaño de la lista, lo cual genera un identificador único para cada canal sin la necesidad de requerir que el programador nombre los canales.

```
new :: SList ss l => Session t ss (ss:>Bot) (Channel t l)
```

La dualidad también se expresa a través de dependencias funcionales y *constraints*, en el listado 19 mostramos las definiciones recursivas para `Send` y `Recv`.

5.3.1. Integración con LiquidHaskell

En principio parece ser posible utilizar LiquidHaskell en conjunto con esta implementación utilizando tipos etiquetados, como podemos ver en el listado 20, en este caso el contrato sería que el primer valor enviado por `server` debe ser distinto de 0. El uso de `liquidAssert` en este caso es puramente demostrativo, no es necesario para la correctitud. Lo usamos simplemente para mostrar que `client` puede asumir que vale el contrato.

En cuanto a usar *refinamientos inline*, el principal problema con el que nos encontramos es que los tipos son particularmente complicados de leer y escribir, como se muestra en listado 21. Además, incluso si ignoramos esto, debido a que los tipos son parte del *constraint* y no de los argumentos, no es posible agregar refinamientos directamente sobre el tipo, ya que estos son directamente ignorados por LiquidHaskell.

```
class Dual n s t | s -> t, t -> s where
  dual :: n -> IO (s, t)

instance (Dual n u u', t ~ t') => Dual n (Send t u) (Recv t' u') where
  dual n = do
    m <- newEmptyMVar
    (u, u') <- dual n
    return (Send (putMVar m) u, Recv (takeMVar m) u')
instance (Dual n u' u, t ~ t') => Dual n (Recv t' u') (Send t u) where
  dual n = do
    m <- newEmptyMVar
    (u, u') <- dual n
    return (Recv (takeMVar m) u, Send (putMVar m) u')
```

Listing 19: Definición de dualidad mediante dependencias funcionales

```
data NonZero = NZ Int deriving (Show)
{-@ data NonZero = NZ {v:Int | v > 0} @-}

server c = do
      x <- recv c
      send c (NZ 1) -- SAFE, pero 0 o x no
      send c 1
      y <- recv c
      return ()

client c = do
   send c 0
   NZ x <- recv c
   return $ liquidAssert (x > 0)
   y <- recv c
   send c (y+1)
   return ()

{-@ liquidAssert :: {b:Bool | b} -> () @-}
liquidAssert _ = ()
```

Listing 20: Ejemplo integración mediante un tipo etiquetado

```
client
  :: (SList tt1 l1, SList tt2 l2, SList ss l3, SList tt3 l4,
      PickupR tt1 (SubT l1 (S n)) (Send v1 a1),
      PickupR tt2 (SubT l2 (S n)) (Recv NonZero u1),
      PickupR ss (SubT l3 (S n)) (Send v2 a2),
      PickupR tt3 (SubT l4 (S n)) (Recv v1 u2), Reify (SubT l1 (S n)),
      Reify (SubT l2 (S n)), Reify (SubT l3 (S n)),
      Reify (SubT l4 (S n)), UpdateR tt3 (SubT l4 (S n)) u2 tt1,
      UpdateR ss (SubT l3 (S n)) a2 tt2,
      UpdateR tt2 (SubT l2 (S n)) u1 tt3,
      UpdateR tt1 (SubT l1 (S n)) a1 uu, IsEnded tt3 F, IsEnded ss F,
      IsEnded tt2 F, IsEnded tt1 F, Num v2, Num v1) =>
     Channel t n -> Session t ss uu ()
```

Listing 21: Tipo de client

5.4. Priority sesh

En [KD21] se propone una implementación de tipos de sesión con linearidad haciendo uso de la extensión de **LinearHaskell**, incluida en GHC 9, lo cual permite representar la estructura de la sesión mediante tipos recursivos relativamente simples, a diferencia de las implementaciones que utilizan el sistema de tipos de Haskell (con sus diversas extensiones) para garantizar el uso lineal de los canales.

5.4.1. Tipos de sesión

Para la representación de los tipos de sesión, en este caso se usan simplemente tipos paramétricos recursivos, y el uso lineal está enforzado por las primitivas que actúan sobre estos, que van desarmando la estructura (a diferencia de las vistas en los apartados 5.2 y 5.3, donde las primitivas arman la estructura de afuera hacia adentro):

```
newtype Send a s = Send (OneShot.SendOnce (a, Dual s))
newtype Recv a s = Recv (OneShot.RecvOnce (a, s))
newtype End      = End OneShot.SyncOnce

send :: Session s => (a, Send a s) %1 -> Linear.IO s
recv :: Recv a s %1 -> Linear.IO (a, s)
close :: End %1 -> Linear.IO ()
```

Listing 22: Send y Recv en priority-sesh

La dualidad se implementa a través de la clase `Session` mediante una dependencia funcional, expresada a través de una *type family* asociada, a diferencia del método usado en apartado 5.3, donde se utilizan clases de varios parámetros. La definición en Haskell se muestra en el listado 23, donde podemos notar que se expresan varios requerimientos sobre el tipo de sesión. `Consumable` es una clase definida en *linear-base* cuyo rol es similar al de un destructor, puede usarse para descartar un valor de tipo lineal, mediante la función: `consume :: a %1 -> ()`.

```
class ( Consumable s
      , Session (Dual s)
      , Dual (Dual s) ~ s
      ) => Session s where
  type Dual s = result | result -> s
  new :: Linear.IO (s, Dual s)
```

Listing 23: Dual en priority sesh

`Session (Dual s)` significa que el dual de un tipo de sesión tiene que ser también un tipo de sesión. `Dual(Dual s)` requiere que la relación de dualidad sea involutiva (es decir, que es su propia inversa). Finalmente, `new` es utilizada para construir un par de canales de tipos duales.

5.4.2. Integración con LiquidHaskell

Si bien LiquidHaskell no tiene en cuenta la extensión de tipos lineares, las puede interpretar como funciones normales, por lo cual no hay problema en usar las dos cosas en conjunto. Aun así, hay ciertas inconveniencias que hacen que no sea trivial de integrar, y que vale la pena comentar. En primer lugar, actualmente [3] LiquidHaskell no compila con GHC 9, debido a un problema con la anotación de `ForeignPtr.withForeignPtr` en *liquid-base*, aunque esto puede salvarse descartando ese refinamiento en particular, ya que esta función no es requerida por *priority-sesh*.

Otro problema es que el constraint de `Session (Dual s)` en `Session` causa un stack overflow cuando se compila con LiquidHaskell.

Por último, LiquidHaskell tiene anotaciones para *base* provistas a través de *liquid-base*, pero no para *linear-base*, por lo cual es necesario anotar los equivalentes lineares sobre los que se quiera predicar.

5.5. Resultados

Como podemos ver en el cuadro 5.1, no es viable escribir refinamientos inline sobre los tipos de la sesión en los primeros dos casos, ya que los tipos a enviar y recibir están embebidos en el sistema de efectos, y no son tipos que LiquidHaskell entienda. En cuanto al tercer caso, si bien es posible refinar los tipos a enviar/recibir, esto solo puede hacerse de un lado de la sesión. La razón es que, como se muestra en cuadro 5.2, se utiliza una *type family* para computar el tipo dual, y como no hay forma de definir los refinamientos de este, LiquidHaskell asume el tipo más general. En el listado 24 podemos ver un ejemplo que ilustra el problema con utilizar una *type family*, en este caso `Id` esencialmente borra los refinamientos, ya que LiquidHaskell no tiene conocimiento sobre el resultado de la evaluación de la "función" (la *type family*), ya que no es capaz de razonar sobre su implementación. A consecuencia de esto, la función `id` es rechazada, y el ejemplo falla en compilar, aunque es evidente que es correcto.

[3] Hasta el commit: 99bdb89.

Implementación	Refinamientos inline	Etiquetas
Effect-sessions	no	no (unsound)
Full-sessions	no	sí
Priority-sesh	parcial	sí

Tab. 5.1: Compatibilidad con LiquidHaskell

Implementación	Dualidad	Tipos lineares
Effect-sessions	TypeFamilies	TypeFamilies
Full-sessions	FunctionalDependencies	FunctionalDependencies
Priority-sesh	TypeFamilies&FunctionalDependencies	LinearHaskell

Tab. 5.2: Mecanismos utilizados por cada implementación

```
type family Id (a :: *) :: * where
    Id a = a

id :: a -> Id a
id a = a

-- Esto falla la validación (Liquid Type Mismatch)
{-@ t :: {i:Int | i = 2} @-}
t :: Int
t = id 2
```

Listing 24: Refinamientos perdidos al aplicar type family.

6. CONTRATOS DEPENDIENTES

6.1. Introducción

En capítulos previos vimos brevemente cómo se pueden definir contratos no dependientes, o insensitivos a la historia, utilizando valores etiquetados. A continuación presentamos una posible técnica para definir contratos dependientes sobre una sesión utilizando LiquidHaskell. Para esto decidimos partir de una implementación basada en *priority-sesh*, principalmente porque los tipos son más simples para trabajar con LiquidHaskell. Para poder utilizar las anotaciones de *liquid-base* utilizamos una versión sin tipos lineales, ya que la linearidad es ortogonal a lo que nos interesa mostrar. Aun así, es posible reincorporarlos sin mayores cambios a los contratos para tener una implementación completa.

6.2. Mónadas refinadas

Como mencionamos brevemente en el apartado 2.4, es posible utilizar refinamientos acotados para poder definir la composición de funciones, y en particular, la composición de mónadas (*Kleisli arrows*). En [VBJ15] se presenta una implementación de una mónada, llamada `RIO`, que está definida con una estructura análoga a la de la mónada *State*, pero refinada con dos predicados que se utilizan como precondición y poscondición de manera similar a la utilizada en las mónadas parametrizadas que vimos anteriormente. *State* es una mónada incluida en base que provee un contexto con un estado global de lectura y escritura a un cómputo, definida como:

```
newtype State s a = State {runState :: s -> (s, a)}
```

La interpretación del tipo anterior es que s es el tipo del estado que se utiliza en el cómputo, y a es el tipo de retorno. La función interna se puede interpretar como una función que toma un estado inicial, y "muta" el estado retornando el próximo estado, y un valor a retornar. La implementación original de `RIO` es esencialmente la misma, con la principal diferencia que el estado no es polimórfico, sino que es un tipo específico, que en nuestro caso es simplemente `data World = W`, porque el "estado" que vamos a usar existe solo en la lógica, y lo usaremos para describir efectos. En este trabajo, utilizaremos una versión modificada como se muestra en listado 25, donde introducimos `IO` internamente para poder interactuar fácilmente con canales. Para el lector familiarizado con la noción de monad transformers, se puede pensar que nuestra instancia es análoga a `StateT IO World a`.

```
{-@ data RIO a <p :: World -> Bool, q :: World -> a -> World -> Bool>
      = RIO (runState :: (x:World<p>
        -> IO (a, World)<\w -> {v:World<q x w> | true}>))
  @-}
data RIO a  = RIO {runState :: World -> IO (a, World)}
```

Listing 25: Extensión de RIO con IO

El predicado **p**, se usa para especificar una condición sobre el estado previo; mientras que el predicado **q**, se utiliza para establecer una relación entre el estado previo, el valor retornado y el siguiente estado.

6.3. Contratos mediante efectos

Para describir los efectos de las operaciones de la sesión, podemos utilizar la directiva *assume* en conjunto con *measures* abstractas, como vimos en el apartado 2.2. Luego, para hablar de dependencias, podemos refinar las primitivas agregando transiciones de estado en las poscondiciones (que son meramente descriptivas), y utilizar las precondiciones para enforzar el cumplimiento del contrato. A modo de ejemplo, en el listado 26 vemos una primera aproximación a un contrato dependiente sobre una sesión. La interpretación es que *adder* recibe dos valores y luego envía la suma sobre el mismo canal.

```
{-@ adder :: (Recv Int (Recv Int (Send Int End)))
      -> RIO <{\w -> sum w = 0}> () @-}
adder s = do
    (x, s) <- recvInt s
    (y, s) <- recvInt s
    s <- sendSum (x + y) s
    close s
```

Listing 26: Aproximación a un contrato mediante RIO

Podemos ver que el código de `adder` utiliza las primitivas `sendSum` y `recvInt`. Estas son especializaciones de `send` y `recv`, como podemos observar en su tipo en el listado 27.

```
{-@ assume sendSum :: Session s => v:Int
     -> Send Int s
     -> RIO<{\w -> v = sum w}, {\w1 b w2 -> w2 = w1}> s @-}
sendSum = send

{-@ assume recvInt :: Recv Int s
     -> RIO<{\w -> true},
            {\w1 v w2 -> sum w2 = sum w1 + (fst v)}> (Int, s) @-}
recvInt = recv
```

Listing 27: Aproximación a un contrato mediante RIO

Notar que en este caso las dos operan sobre `Int` en lugar de `a`. Es decir, a diferencia de las operaciones sin contratos, estas no son polimórficas en el valor a recibir/enviar, ya que necesitamos poder sumar.

La idea intuitiva del refinamiento de `recvInt`, es que la variable *sum* se incrementa con el valor recibido. Esta "variable" la representamos mediante una *measure* sobre *World*:

```
{-@ measure sum :: World -> Int @-}
```

Por otro lado, `sendSum` solo puede usarse para enviar un valor que coincida con el valor actual de *sum* que, como solo se modifica mediante `recvInt`, implica que solo puede enviarse un valor que coincida con la suma de los valores recibidos. A continuación

```
data Send tag a s = Send (SendOnce (a, Dual s))
data Recv tag a s = Recv (RecvOnce (a, s))
data End      = End SyncOnce

instance Session s => Session (Send tag a s) where
  type Dual (Send tag a s) = Recv tag a (Dual s)
  newS = bimap Send Recv <$> new'

instance Session s => Session (Recv tag a s) where
  type Dual (Recv tag a s) = Send tag a (Dual s)
  newS = bimap Recv Send . swap <$> new'
```

Listing 28: Extensión con tags de tipos de sesión

```
data Any
data Sum

{-@ assume sendSum :: Session s => v1:Int
     -> Send Sum Int s
     -> RIO<{\w -> v1 = sum w}, {\w1 b w2 -> w2 = w1}> s @-}
{-@ assume recvInt :: Recv Any Int s
     -> RIO<{\w -> true},
            {\w1 v w2 -> sum w2 = sum w1 + (fst v)}> (Int, s) @-}
adderService :: (Recv Any Int (Recv Any Int (Send Sum Int End))) -> RIO ()
```

Listing 29: Extensión de las especializaciones con tags

damos una descripción más detallada de sus refinamientos: El efecto de la comunicación está capturado en los predicados que instancian *RIO*, que son una pre y poscondición, respectivamente. En el caso de `sendSum`, la precondición es `v = sum w` , que requiere que el valor a enviar (**v**) sea igual a la suma. Mientras que la poscondición `w1 = w2` solo indica que el estado no cambia. En el caso de `recvInt`, la precondición es siempre verdadera, lo cual significa que puede llamarse siempre que la sesión tenga tipo `Recv Int`. La poscondición `sum w2 = sum w1 + (fst v)` dice que a partir de un estado de entrada **w1**, que tiene acumulada la suma de valores recibidos: **sum w1**, la suma de valores recibidos del estado siguiente (**w2**) se produce sumando el valor recibido a través del canal: **fst v**, y la suma del estado de entrada. Vale la pena aclarar que si tuviéramos otras variables de estado, también tenemos que aclarar cómo cambian (o si no lo hacen). Es fácil notar que esta implementación tiene un gran problema: no hay nada que fuerce a utilizar `recvInt` ni `sendSum`, dado que el contrato no es parte de los tipos, si no que está implícito por las operaciones que se utilizan.

Para remediar esto, extendemos los tipos de sesión con una etiqueta, que luego asociamos a tanto una instancia de *send*, como a una de *recv*; para permitir el uso de manera dual: listado 28. Además, podemos modificamos los tipos de las operaciones para que contemplen estas etiquetas, como podemos ver en el listado 29.

Además, también podemos definir ahora los tipos necesarios para describir el otro lado de la sesión, como podemos ver en el listado 30.

Vale la pena resaltar que en el caso de los valores recibidos, como en `recvSum`, el

```
{-@ assume sendInt :: Session s => v:Int
      -> Send Sum Int s
      -> RIO<{\w -> true}, {\w1 b w2 -> sum w2 = (sum w1) v}> s @-}
{-@ assume recvSum :: Recv Sum Int s
      -> RIO<{\w -> true}, {\w1 v w2 -> (fst v) = sum w}> (Int, s) @-}
adderUser :: (Send Any Int (Send Any Int (Recv Sum Int End))) -> RIO ()
adderUser s = do
    s <- sendInt 1 s
    s <- sendInt 2 s
    (r, s) <- recvSum s
    close s
```

Listing 30: Usuario del servicio del listado 28.

predicado es meramente asumido, puesto que es parte de la poscondición. Esto significa que la correctitud está subordinada al hecho de que la instancia de la primitiva dual de tiene el mismo predicado, puesto que cada función se verifica por separado. Aunque podría utilizarse algún tipo de mecanización para hacer el proceso de definirlas menos propenso a errores, queda fuera del alcance de este trabajo; en lugar de eso, cuando sea necesario usaremos predicados auxiliares para reducir la repetición de código.

También es posible utilizar el contrato anterior de manera recursiva, ya que la cantidad de valores a sumar no es parte de los refinamientos. El código presentado en listado 31 es un ejemplo de esto: es un proceso que recibe valores indefinidamente, hasta que el usuario del servicio decide que quiere el resultado, que en este caso es la suma de los valores recibidos. Notar que en este caso necesitamos el invariante de que el argumento *tot* tiene la suma actual para poder probarlo, lo cual también fuerza a que en la primera iteración tengamos que pasarle 0. Cabe destacar que para poder verificar este ejemplo es necesario usar el flag de `--no-termination`, ya que por defecto LiquidHaskell valida que las funciones recursivas reduzcan sus argumentos en cada paso, pero si analizamos `sumSrv` por sí sola, no hay garantía de que alguna vez termine, ya que la rama de la derecha podría no tomarse.

6.4. Limitaciones

Una de las principales limitaciones del mecanismo que acabamos de describir, es que es necesario definir manualmente las especializaciones de `send` y `recv`, y cómo cambia el "estado" tras la ejecución de cada una. En particular, lo más problemático es que se tienen que definir tanto una función para enviar como una para recibir, donde ambas tienen que tener el mismo predicado; si por error omitiéramos la validación en la función que envía, entonces estaríamos asumiendo cosas que no necesariamente valen. Potencialmente esto puede resolverse utilizando algún mecanismo de generación de código, como **Template Haskell**, pero no se exploró ninguna de estas posibilidades.

Por otro lado, hay ciertos casos patológicos en los que la cantidad de instrucciones generadas para ser resueltas por el *SMT solver* crece de manera exagerada en función de la cantidad de operaciones en una expresión. Esto parece estar asociado principalmente a la manera en la que se representan los tipos de las primitivas de selección de ramas en esta implementación de tipos de sesión, particularmente las primitivas de `selectLeft`

```
{-@ predicate SumT W1 V W2 = sum W2 = V + (sum W1) @-}
{-@ predicate SumP W V = V = sum W @-}

{-@ assume sendSum :: Session s => v1:Int
     -> Send Sum {v2:Int | v1 = v2} s
     -> RIO<{\w -> SumP w v1}, {\w1 b w2 -> SumT w1 v1 w2}> s @-}
{-@ assume recvAny :: Recv Any Int s
     -> RIO<{\w -> true}, {\w1 v w2 -> SumT w1 (fst v) w2}> (Int, s) @-}

newtype SumSrv
  = MkSumSrv (Offer (Recv Any Int SumSrv) (Send Sum Int End))

{-@ sumSrv :: tot:Int -> SumSrv -> RIO<{\w -> tot = sum w}> () @-}
sumSrv tot (MkSumSrv s) = offerEither s $ \x -> case x of
  Left   s -> do (x, s) <- recvAny s; sumSrv (tot + x) s
  Right  s -> do s <- sendSum tot s; close s
```

Listing 31: Actor recursivo que computa la sumatoria

y `selectRight`. En el listado 32 damos un ejemplo que ilustra el caso problemático en cuestión.

El tipo de la sesión dice que se envían tantos valores enteros como se quiera, hasta que se decide tomar la otra rama, en cuyo caso se espera una respuesta (también un entero), y la sesión termina:

```
newtype UniqCnt
  = MkUniqCnt (Select (Send Uniq Int UniqCnt) (Recv Any Int End))
```

Por otro lado, los refinamientos dicen que solo se pueden enviar valores que no se hayan enviado anteriormente. Esto último se logra utilizando un conjunto como *measure*, y agregando a este cada valor enviado como parte de la poscondición de `sendUniq`, mediante el predicado auxiliar `SendT`; y utilizando `UniqP` para enforzar la propiedad:

```
{-@ predicate SentT W1 V W2 = sent W2 = Set_cup (Set_sng V) (sent W1) @-}
{-@ predicate UniqP W V = not (Set_mem V (sent W)) @-}

{-@ assume sendUniq :: Session s => v1:Int -> Send Uniq Int s ->
     RIO<{\w -> UniqP w v1},
         {\w1 b w2 -> SentT w1 v1 w2}> s @-}
```

Si bien LiquidHaskell puede validar correctamente que se cumple el contrato, simplemente repetir las operaciones de `selectLeft` y `sendUniq` causa que el archivo con las instrucciones de SMT crezca desmesuradamente. Por ejemplo, como ilustramos en el listado 32, el tamaño del archivo que se pasa al *solver* pasa de 160K a 372M al enviar 5 valores.

En cuanto a soluciones alternativas, en principio es posible evitar este problema utilizando funciones más chicas, aunque esto requiere de anotar todos los predicados intermedios, lo cual puede ser redundante. En este caso en particular, otra opción es escribirlo

```
{-@ uniqCnt :: UniqCnt -> RIO <{\w -> Set_emp (sent W)}> () @-}
uniqCnt (MkUniqCnt s) = do
    s <- selectLeft s
    MkUniqCnt s <- sendUniq 3 s
    -- 160K
    s <- selectLeft s
    MkUniqCnt s <- sendUniq 4 s
    -- 628K
    s <- selectLeft s
    MkUniqCnt s <- sendUniq 5 s
    -- 4.7M
    s <- selectLeft s
    MkUniqCnt s <- sendUniq 1 s
    -- 42M
    s <- selectLeft s
    MkUniqCnt s <- sendUniq 2 s
    s <- selectRight s
    -- 372M
    (_, s) <- recvAny s
    close s
```

Listing 32: Actor que envía valores sin repetidos

```
{-@ type UList a = {v:[a] | Set_emp (dups v)}     @-}

{-@ uniqCnt :: xs:UList Int -> UniqCnt ->
     RIO<{\w -> not (Set_mem (head xs) (sent w))}> () @-}
uniqCnt (x:xs) (MkUniqCnt s) = do
    s <- selectLeft s
    s <- sendUniq x s
    nop
    uniqCnt xs s
uniqCnt [] (MkUniqCnt s) = do
    s <- selectRight s
    (x, s) <- recvAny s
    close s

{-@ nop :: RIO<{\w -> true}> () @-}
nop = return ()
```

Listing 33: Solución al problema ilustrado en listado 32

como una función recursiva sobre una lista, como mostramos en el listado 33. En este caso, *dups* es una measure que retorna el conjunto de elementos que aparecen más de una vez en una lista, omitimos su implementación por brevedad, pero es fácil de definir mediante recursión explícita. `uniqCnt` en este caso toma una lista sin elementos repetidos como primer argumento, por lo cual sabemos que podemos enviarlos todos directamente; sin esta precondición, tendríamos que antes pre-procesar la lista o mantener un conjunto de valores enviados. Como en el caso de `sumSrv` que mostramos anteriormente, también necesitamos un invariante que relaciona el primer argumento con el estado: en este caso la precondición de RIO dice que el primer elemento de la lista no se encuentra en la lista de elementos ya enviados. Sin embargo, como podemos notar, esto tampoco funciona perfectamente, ya que por razones desconocidas LiquidHaskell no puede probar el ejemplo si no ponemos alguna acción antes de la llamada recursiva, como se puede ver en este caso con la función *nop* en la línea 8. Como su nombre lo indica, y como se puede ver en su implementación, esta función no tiene ningún efecto, pero sin ella el ejemplo es **UNSAFE**.

7. CONCLUSIONES

Como vimos al comienzo, el uso de tipos lineales es fundamental para garantizar el uso correcto de los tipos de sesión. Esto significa que las implementaciones pensadas para versiones anteriores a GHC 9 tienen que implementar su propio sistema de tipos lineales. Desafortunadamente, los mecanismos que pueden usarse para esto no son particularmente compatibles con los tipos de LiquidHaskell. Por otro lado, incluso si se cuenta con un sistema de tipos lineales como en GHC 9, los mismos mecanismos también se requieren para poder implementar la relación de dualidad, por lo cual de todas formas se dificulta escribir refinamientos sobre el tipo de sesión. En principio, si se quiere escribir solo contratos insensitivos a la historia, utilizar tipos etiquetados con refinamientos sobre los constructores tiene un costo integración relativamente bajo. Si se requiere de escribir contratos con dependencias sobre los valores anteriores, utilizar mónadas refinadas como vimos en el capítulo anterior es un mecanismo viable para escribir contratos sobre objetos mutables, o con efectos, como lo son las sesiones. En principio, esta técnica parece ser combinable con cualquier representación de la sesión, ya que los refinamientos están en la mónada y no en el tipo de sesión en sí. Sin embargo, encontramos al menos un caso en el cual LiquidHaskell nos permite probar cosas que no son verdad cuando se combina con ciertas extensiones del lenguaje, por lo cual es recomendable analizar con detenimiento la implementación con la cual se va a integrar. Además, en el caso de las mónadas refinadas, encontramos ciertos casos en los cuales podemos tener importantes problemas de performance. Si bien es posible atenuar estos mediante anotaciones intermedias y reescrituras, esto requiere intervención del programador para adaptar la implementación al sistema de pruebas, lo cual puede ser no deseable en todos los casos; además de que es ligeramente contrario al propósito de usar una herramienta de razonamiento automático como LiquidHaskell.

BIBLIOGRAFíA

[Mey92] B. Meyer. «Applying 'design by contract'». En: *Computer* 25.10 (1992), págs. 40-51. DOI: 10.1109/2.161279.

[Hon93] Kohei Honda. «Types for dyadic interaction». En: *CONCUR'93*. Ed. por Eike Best. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, págs. 509-523. ISBN: 978-3-540-47968-0.

[KHB99] Murat Karaorman, Urs Hölzle y John Bruno. «jContractor: A Reflective Java Library to Support Design By Contract». En: *Meta-Level Architectures and Reflection*. Ed. por Pierre Cointe. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, págs. 175-196. ISBN: 978-3-540-48443-1.

[Aic03] Bernhard K. Aichernig. «Contract-Based Testing». En: *Formal Methods at the Crossroads. From Panacea to Foundational Support: 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002. Revised Papers*. Ed. por Bernhard K. Aichernig y Tom Maibaum. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, págs. 34-48. ISBN: 978-3-540-40007-3. DOI: 10.1007/978-3-540-40007-3_3. URL: https://doi.org/10.1007/978-3-540-40007-3_3.

[Sul+06] Martin Sulzmann y col. «Understanding Functional Dependencies via Constraint Handling Rules». En: *Journal of Functional Programming* 17 (ene. de 2006). DOI: 10.1017/S0956796806006137.

[GF07] Jessica A. Gronski y Cormac Flanagan. «Unifying Hybrid Types and Contracts». En: *Trends in Functional Programming*. 2007.

[Pey08] Simon Peyton Jones. «Static Contract Checking for Haskell». En: *POPL'09*. University of Cambridge. ACM, ago. de 2008, págs. 41-52. URL: https://www.microsoft.com/en-us/research/publication/static-contract-checking-for-haskell/.

[RKJ08] Patrick M. Rondon, Ming Kawaguci y Ranjit Jhala. «Liquid Types». En: *SIGPLAN Not.* 43.6 (jun. de 2008), págs. 159-169. ISSN: 0362-1340. DOI: 10.1145/1379022.1375602. URL: https://doi.org/10.1145/1379022.1375602.

[PT09] Riccardo Pucella y Jesse A. Tov. «Haskell session types with (almost) no class». English (US). En: *ACM SIGPLAN Notices* 44.2 (feb. de 2009), págs. 25-36. ISSN: 1523-2867. DOI: 10.1145/1543134.1411290.

[Boc+10] Laura Bocchi y col. «A Theory of Design-by-Contract for Distributed Multiparty Interactions». En: *Proceedings of the 21st International Conference on Concurrency Theory*. CONCUR'10. Paris, France: Springer-Verlag, 2010, págs. 162-176. ISBN: 3642153747.

[HT10] Phillip Heidegger y Peter Thiemann. «Contract-driven testing of JavaScript code». En: *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer. 2010, págs. 154-172.

[IYA11] Keigo Imai, Shoji Yuen y Kiyoshi Agusa. «Session Type Inference in Haskell». En: *PLACES* 69 (oct. de 2011). DOI: 10.4204/EPTCS.69.6.

[Vas12] Vasco T. Vasconcelos. «Fundamentals of session types». En: *Information and Computation* 217 (2012), págs. 52-70. ISSN: 0890-5401. DOI: https://doi.org/10.1016/j.ic.2012.05.002. URL: https://www.sciencedirect.com/science/article/pii/S0890540112001022.

[GG13] Dennis Griffith y Elsa Gunter. «LiquidPi: Inferrable Dependent Session Types». En: vol. 7871. Mayo de 2013, págs. 185-197. DOI: 10.1007/978-3-642-38088-4_13.

[Vaz+14] Niki Vazou y col. «Refinement Types for Haskell». En: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. ACM, sep. de 2014, págs. 269-282. ISBN: 978-1-4503-2873-9. URL: https://www.microsoft.com/en-us/research/publication/refinement-types-for-haskell/.

[VBJ15] Niki Vazou, Alexander Bakst y Ranjit Jhala. «Bounded Refinement Types». En: *ACM SIGPLAN Notices* 50 (jul. de 2015). DOI: 10.1145/2858949.2784745.

[HYC16] Kohei Honda, Nobuko Yoshida y Marco Carbone. «Multiparty Asynchronous Session Types». En: *J. ACM* 63.1 (mar. de 2016). ISSN: 0004-5411. DOI: 10.1145/2827695. URL: https://doi.org/10.1145/2827695.

[OY16] Dominic Orchard y Nobuko Yoshida. «Effects as Sessions, Sessions as Effects». En: *SIGPLAN Not.* 51.1 (ene. de 2016), págs. 568-581. ISSN: 0362-1340. DOI: 10.1145/2914770.2837634. URL: https://doi.org/10.1145/2914770.2837634.

[Ber+17] Jean-Philippe Bernardy y col. «Linear Haskell: practical linearity in a higher-order polymorphic language». En: *CoRR* abs/1710.09756 (2017). arXiv: 1710.09756. URL: http://arxiv.org/abs/1710.09756.

[GR17] S. Gay y A. Ravara. *Behavioural Types: from Theory to Tools*. River Publishers Series in Automation, Control and Robotics. River Publishers, 2017. ISBN: 9788793519824. URL: https://books.google.com.ar/books?id=6x0vDwAAQBAJ.

[MP17] Hernán Melgratti y Luca Padovani. «Chaperone Contracts for Higher-Order Sessions». En: *Proc. ACM Program. Lang.* 1.ICFP (ago. de 2017). DOI: 10.1145/3110279. URL: https://doi.org/10.1145/3110279.

[TY18] Bernardo Toninho y Nobuko Yoshida. *Depending on Session-Typed Processes*. 2018. DOI: 10.48550/ARXIV.1801.08114. URL: https://arxiv.org/abs/1801.08114.

[KD21] Wen Kokke y Ornela Dardha. «Deadlock-Free Session Types in Linear Haskell». En: *CoRR* abs/2103.14481 (2021). arXiv: 2103.14481. URL: https://arxiv.org/abs/2103.14481.