VNIVERSIDAD NACIONAL AVTNOMA DE MEXICO

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

Programación dinámica puramente funcional: el caso de la *memoización*

T E S I S

QUE PARA OBTENER EL GRADO DE:

MAESTRA EN CIENCIAS
(COMPUTACIÓN)

P R E S E N T A :

LOURDES DEL CARMEN GONZÁLEZ HUESCA

DIRECTORES DE TESIS

DR. FRANCISCO HERNÁNDEZ QUIROZ
DR. FAVIO EZEQUIEL MIRANDA PEREA

México D.F. 2010

Agradecimientos [1]

Gracias a ti, que me has apoyado siempre aunque no lo dices, que me has cuidado desde que sabías de mi existencia y que siempre te has preocupado por mi.

Gracias a ti, que me cuidaste y amaste como a tu hija, por hacerme comprender que todo tiene solución menos la muerte.

Gracias a ti, que me educaste y cuidaste, que me enseñaste a tener paciencia.

Gracias a ti, que me hiciste responsable y me enseñaste a ser la hermana mayor.

Gracias a ti, que me hiciste compañia y me permitiste tener una hermana.

Gracias a ti, que me has acompañado en mis estudios y siempre has sido un apoyo incondicional.

Gracias a todos lo que han sido parte de mi camino en la UPAEP pero sobre todo a mis compañeros de clase y a los que se volvieron mis amigos, a los profesores que me dieron clase y compartieron sus conocimientos, a mis compañeros de trabajo y a mis alumnos por enseñarme a enseñar, a Liliana y a Favio por confiar en mi y adentrarme más en el reino de la lógica.

[1] Estos agradecimientos valen doble.

Índice general

Introducción

La *memoización*[2] es una técnica fundamental para evitar la redundancia computacional. Una función *memoizada*[3] se parece a una función ordinaria excepto que almacena en una tabla sus valores previamente computados. Es decir, si una función memoizada f es aplicada por segunda vez a un argumento particular, devuelve inmediatamente el resultado almacenado en la tabla en lugar de calcularlo nuevamente. De esta manera esta técnica evita la evaluación o el cómputo repetido ejecutando en su lugar una búsqueda en la tabla de valores almacenados.

La programación dinámica resuelve problemas al combinar soluciones de subproblemas. Esta idea es parecida a la ocupada por el principio de *divide y vencerás*, el cual separa al problema en subproblemas independientes que se resuelven recursivamente y cuyas soluciones se combinan para obtener la solución del problema original. La diferencia entre la técnica de memoización y la programación dinámica es que en la primera los subproblemas no son independientes, es decir se comparten subsubproblemas como está descrito en [5]. Las subsoluciones encontradas se almacenan en una tabla para evitar cómputos que ya se habían realizado. La memoización de funciones o algoritmos proporciona una variación de la programación dinámica al mantener una tabla de valores previos sin modificar un algoritmo recursivo, que es natural aunque ineficiente. Ejemplos clásicos de programación dinámica, como el problema de la subsecuencia común más larga o el problema del costo de la multiplicación de matrices, se sirven de manera importante de esta técnica.

Una función memoizada así como su tabla de valores pueden ser implementadas de diversas maneras, en particular la tabla puede ser implementada vía tablas dispersas o *tablas hash*, árboles de búsqueda basados en comparaciones, etc. De misma forma se pueden utilizar diversos paradigmas de programación para aprovechar el reuso de resultados. En nuestro caso se busca la definición e implementación de esta técnica en el ámbito de la programación funcional pura. Si bien a primera vista la ausencia de efectos laterales tales como la asignación parecen un obstáculo en la implementación en el ámbito mencionado, se debe notar que las únicas partes de un programa que deben ser memoizadas son aquellas puramente funcionales, es decir, aquellas en las que el resultado dependa únicamente de los argumentos de entrada y en las que no hay efectos laterales. La razón para esto es que el proceso de búsqueda en la tabla utiliza los argumentos de entrada como "llaves", de manera que si el resultado final

[2]En inglés *Memoization*
[3]*memoized*

depende de otros valores accesibles mediante llamadas que no son puramente funcionales, la tabla devolverá un valor incorrecto. Adicionalmente si un componente de programa tiene efectos laterales, como por ejemplo incrementar un contador, la memoización podría destruirlos o ignorarlos. En conclusión una condición para una correcta implementación de esta técnica es el uso de un lenguaje funcional puro que permita el manejo de las funciones y los posibles efectos laterales que sucedan durante el cómputo.

Por otra parte hay que asegurar que la memoización no modifique los resultados de un programa ni mucho menos alguna propiedad de terminación del programa, es decir que los resultados y propiedades de un programa que no está memoizado deben ser los mismos que resulten si se memoizara. Determinar las partes puramente funcionales de un programa para memoizarlas puede ser muy difícil, es mejor desarrollar todo en un ambiente puramente funcional. Pero en la programación funcional pura no se pueden implementar a las tablas como se haría en un lenguaje estructurado o imperativo, no pueden ser desarrolladas como objetos de almacenamiento que puedan actualizarse, esto es que no requieran rehacerse. De ahí que sean necesarias otras técnicas para desarrollar la técnica de memoización en el ámbito funcional.

En este trabajo de tesis se explorarán diferentes enfoques de implementación de la técnica de memoización, su contenido está organizado de la siguiente forma: el primer capítulo incluye conceptos básicos relacionados con todo el trabajo, es decir, contiene las bases para entender los enfoques a explorar además de explicar el paradigma de programación funcional pura. El segundo capítulo está dedicado a discutir la memoización además de revisar dos de los enfoques que se han ocupado para implementarla: mónadas y programación politípica. El tercer capítulo está dedicado a la memoización selectiva, se presenta una modificación del sistema expuesto en [2] así como una demostración de la seguridad de ese sistema. Finalmente se discuten las conclusiones y el trabajo futuro.

Capítulo 1

Preliminares

Para el desarrollo de esta tesis se necesitarán algunos conocimientos previos como conceptos básicos sobre tipos modales, mónadas y programación politípica así como su manejo y uso dentro de la programación funcional.

1.1. Tipos Modales

Desde los griegos se ha estudiado el razonamiento humano y para ello los sistemas formales lógicos han sido una parte importante de la filosofía y de las ciencias. Estos sistemas formales lógicos proporcionan un lenguaje y una semántica o interpretación que permiten la representación de razonamientos a través de fórmulas bien formadas del lenguaje y la evaluación de ellos mediante un modelo que les asigna un valor, verdadero o falso. Algunos de estos sistemas son: el cálculo de proposiciones, el cálculo de predicados, la lógica modal, etc., con extensiones o restricciones.

Sin embargo, existen razonamientos que involucran tiempo, cambios, acciones o eventos. Por ejemplo la siguiente frase puede cambiar con el tiempo: *un boleto del metro cuesta 2 pesos*, en cambio la afirmación *27 es el cubo de 3* no cambiará con el tiempo. Para reflejar estas diferencias se han desarrollado sistemas que ayudan a formalizar el conocimiento y los eventos que pueden cambiar con el tiempo. El ***cálculo de eventos***, desarrollado por Robert Kowalski y Marek Sergot en 1986, permite expresar acciones y cambios en el tiempo. Usando lógica de primer orden se puede determinar el punto en el tiempo donde sucede alguna acción; las funciones expresan las acciones y los predicados expresan el momento en el tiempo en el que sucede una acción o permiten expresar el inicio o el fin de una acción.

Otros sistemas son las lógicas modales que forman parte de las lógicas no clásicas. Una lógica modal es un sistema formal lógico que maneja modalidades, las cuales califican la veracidad de un juicio. Las dos clases de modalidades más comunes son: necesidad y posibilidad. El lenguaje de la lógica modal propisicional extiende al de la lógica de proposiciones incorporando símbolos para representar las modalidades. Pero en la lógica modal es necesario

hacer una asignación de valores de verdad respecto al tiempo. Así las fórmulas atómicas no podrían tener esta asignación ya que se deben hacer diferencias entre grados de verdad. En 1963, Saul Kripke propuso la semántica de la lógica modal de conocimiento en términos de mundos posibles.

Las lógicas modales han tenido diversas interpretaciones, aquí tomaremos una en donde se presenta en un sistema de deducción natural, útil para interpretarse como un sistema de tipos para lenguajes de programación. Asimismo las aplicaciones que en este trabajo se desarrollan sólo requieren de la modalidad de necesidad la cual se explica a continuación.
Dentro de la semántica de mundos posibles o de Kripke, la modalidad de necesidad ($\Box$), es un operador unario y se interpreta de la siguiente forma:

$$\Box\, p \;\; significa\ que\ p\ siempre\ es\ verdadera\ en\ el\ futuro$$

Esta interpretación puede cambiar dependiendo del contexto en donde es usada la lógica modal, por ejemplo ($\Box A$) puede tener las siguientes lecturas:

- Es necesariamente cierto que A
- Siempre será cierto que A
- Que debe de ser A
- Que se cree que A
- Que se conoce A
- Después de la ejecución de un programa P, se cumple A

La interpretación que nos interesa es la siguiente:

$$la\ proposición\ (\Box A)\ significa\ que\ A\ es\ válida. \tag{1.1}$$

Esta interpretación se entenderá mejor a lo largo de la siguiente subsección.

1.1.1. Juicios, verdad y validez

Martin-Löf hace una separación entre juicios y proposiciones, juzgar es conocer, un juicio evidente es un objeto de conocimiento y una prueba o demostración es lo que hace a un juicio evidente. La lógica se basa en la noción de juicios, así que saber o conocer una proposición significa que se sabe cuál es la verificación de ella.
Los juicios más importantes en lógica son aquellos de la forma *A es verdadera* donde A es una proposión, por tanto se requiere previamente de juicios para proposiciones, es decir de la forma *A es una proposición*. Como se interpreta en [26], saber que *A es una proposición* significa que se conoce lo que verifica a A y saber que *A es verdadera* significa que se sabe cómo verificar A.

La caracterización por medio de juicios permite tener una visión de la demostración, se manejan dos reglas por cada conectivo u operador. Estas reglas son las conocidas como de introducción y de eliminación en los sistemas de deducción natural debido a Gentzen que formuló este tipo de sistemas. El significado de una proposición se obtiene de los objetos que son utilizados para demostrarla. Así las reglas de introducción permiten concluir cuándo las proposiciones son verdaderas y las reglas de eliminación proporcionan una forma de obtener información al destruir o separar una proposición.

Los juicios de las formas *A es una proposición* y *A es verdadera* no son suficientes para demostrar el esquema correspondiente a la implicación, para ello se introducen los juicios hipotéticos como se explica en [26]. Los juicios hipotéticos, $A_1, \ldots, A_n \vdash B$, consideran a las premisas A_i junto con la conclusión B como proposiciones verdaderas además de que las hipótesis se consideran evidencia.
Existen juicios que no dependen de las hipótesis en el sentido de que no importa la veracidad de las proposiciones, estos son los juicios categóricos. Aquí se introduce el juicio para *A es válida* suponiendo que A es una proposición. La evidencia para la validez de A es la evidencia de A:

$$\text{Si } \cdot \vdash (A \textit{ verdadera}) \text{ entonces } A \textit{ válida}$$
$$\text{Si } A \textit{ válida} \text{ entonces } \Gamma \vdash A \textit{ verdadera}$$

Las hipótesis en Γ se consideran información válida. Así el significado para el juicio de validez para proposiciones se explica a través de la noción de verdad.
Ahora se pueden incorporar hipótesis válidas como información dentro de los juicios hipotéticos mediante la colección Δ. El uso de la separación entre proposiciones verdaderas y válidas se hará evidente en el capítulo tres.

A partir de la interpretación dada anteriormente (1.1), se puede derivar el siguiente juicio categórico para la proposición *A es válida*:

$$\frac{A \textit{ prop}}{\Box A \textit{ prop}} \Box F$$

Por tanto tenemos dos reglas de inferencia para la modalidad de necesidad incluyendo las dos colecciones de información:

$$\frac{\Gamma \mid \cdot \vdash A \textit{ verdadera}}{\Gamma \mid \Delta \vdash \Box A \textit{ verdadera}} \Box I$$

$$\frac{\Gamma \mid \Delta \vdash \Box A \textit{ verdadera} \qquad \Gamma,\ A \textit{ válida} \mid \Delta \vdash B \textit{ verdadera}}{\Gamma \mid \Delta \vdash B \textit{ verdadera}} \Box E$$

Este sistema modal lógico es estudiado a fondo en [26] haciendo la distinción entre juicios y proposiciones. El objetivo lógico de este sistema es dar un significado constructivo de la modalidad de necesidad en el sentido intuicionista mediante un sistema de deducción natural.

Así mismo, en [26], al seguir el trabajo de Martin-Löf se discute una distición entre juicios de acuerdo a las evidencias de él:

- Un juicio analítico es aquel que es evidente por si mismo, así el juicio A *prop* es analítico ya que es suficiente el conocimiento de A como evidencia.

- Un juicio sintético es aquel en el que se debe buscar evidencia fuera de él para justificarlo, así el juicio A *verdadera* es sintético ya que se debe buscar una prueba de A como evidencia.

El uso de la lógica en ciencias de la computación es importante ya que las pruebas aportan construcciones y algoritmos. Martin-Löf utiliza los juicios analíticos para incorporar teoría de tipos mediante términos M para indicar que M *es una demostración de* A, con alguna de las siguientes notaciones:

$$M : proof(A) \qquad M : A$$

Esta notación permite reescribir el sistema anterior mediante:

$$\frac{\Gamma \mid \cdot \vdash M : A}{\Gamma \mid \Delta \vdash \mathsf{box}\, M : \Box A}\Box I$$

$$\frac{\Gamma \mid \Delta \vdash M : \Box A \qquad \Gamma,\ u :: A \mid \Delta \vdash N : B}{\Gamma \mid \Delta \vdash \mathsf{let\,box}\ u = M\ \mathsf{in}\ N : B}\Box E$$

La correspondencia entre pruebas-proposiciones y términos-tipos que se aprecia en las dos versiones de un sistema para la modalidad de necesidad es la llamada de Curry-Howard que relaciona a la lógica intuicionista con el cálculo lambda tipado. Este último se revisará en la siguiente sección.

1.2. Programación funcional

La programación funcional es un paradigma de programación en el cual se tratan los cómputos como evaluaciones, se construyen definiciones que sirven como reglas para simplificar expresiones mediante la aplicación de funciones y así resolver problemas. Las funciones son el centro de este paradigma, en el caso extremo no hay estados ni datos mutables y se deben considerar las relaciones entre valores.

La programación funcional se basa en el cálculo lambda, sistema formal desarrollado alrededor de 1930 por Alonzo Church que involucra funciones, definiciones, aplicaciones, recursión, etc. Muchas características de los lenguajes de programación actuales pueden estudiarse mediante extensiones del cálculo lambda.
Hay varias características que hacen al cálculo lambda importante: Church encontró que la noción de un "cómputo efectivo" se puede formalizar dentro de este sistema, el resultado de una función depende sólo de los argumentos de entrada sin tener que depender del estado en el que se ejecuta el programa, así los efectos laterales, producto de la evaluación de una

expresión dependiente de ciertas definiciones de funciones, no son parte central de la programación funcional pura. Otra característica es que el cálculo lambda es un sistema que permite definir funciones de orden superior así como tipos de datos.

De las características nombradas antes la más sobresaliente es la segunda, el resultado de una función sólo depende de sus argumentos de entrada. Gracias a esta característica se obtiene una propiedad primordial de las funciones en este paradigma: la transparencia referencial[1]. También se puede decir que una función transparente referencialmente es una función pura, es decir es una función en el sentido matemático. Esta propiedad indica que la evaluación de una función puede ser ignorada y en su lugar devolver el resultado correspondiente al o los argumentos de entrada de la función, es decir que la evaluación de la función puede ser sustituida en su totalidad por el resultado que se espera. Como consecuecia se puede deducir que no hay efectos laterales durante la evaluación de la función asegurando que el resultado global no se vea afectado si una función no es evaluada y sólo se devuelve el valor esperado. Un ejemplo de funciones determinadas son las funciones matemáticas, éstas no generan efectos laterales. La referencia transparencial es importante ya que permite razonar respecto al comportamiento del programa así se puede probar la correctud del mismo al igual que optimizarlo como lo hace la técnica de memoización.

Una observación importante es que todas las estructuras de datos en la programación funcional son persistentes. Computacionalmente una estructura de datos persistente es aquella que preserva la versión anterior de ella, las operaciones de actualización de la estructura siempre generan una nueva estructura actualizada, es decir que al momento de actualizar una estructura se tienen ambas, la vieja y la nueva estructura. Este tipo de estructuras son inmutables, la ventaja que ofrece la persistencia de las estructuras es que se pueden aprovechar las diferentes versiones de un objeto para su análisis y optimización.

Dentro de los lenguajes enfocados en el paradigma funcional existen diferencias en cuanto a la evaluación de expresiones, conocido como estrategias de evaluación. Hay dos principales estrategias, la estricta y la no estricta, dentro de esta última se encuentra la *perezosa*. El orden en la evalución dentro de los lenguajes perezosos no está especificado, por tanto la evaluación perezosa presenta varias ventajas como evitar la evaluación de argumentos que no son utilizados dentro de otra evaluación, inclusive sólo se pueden examinar las partes necesarias de una estructura sin construir a la estructura completa de ahí que se puedan manipular estructuras infinitas.

Tratar con programación puramente funcional es tratar con programación funcional pero sin modificaciones destructivas o actualizaciones. La técnica de memoización esté presente en la programación funcional pura bajo la evalución perezosa debido a la evaluación de argumentos. Los argumentos se evalúan la primera vez que son requeridos y sus valores se almacenan para reutilizarlos posteriormente, asegurando que un argumento sólo será evaluado una única vez.

[1] *referential transparency* en inglés

HASKELL es un lenguaje puramente funcional basado en el sistema F_ω, extensión polimórfica del cálculo lambda, incluye recursión y manejo de efectos. Su semántica es no estricta y es fuertemente tipado. Sus principales características son las discutidas antes y otras más:

- transparencia referencial o pureza
- evaluación perezosa
- tiene un sistema de tipos basado en el sistema F_ω
- incluye un mecanismo para definir tipos de datos algebraicos
- utiliza el emparejamiento de patrones (*pattern matching*)
- maneja clases de tipos para soportar el polimorfismo
- permite currificar funciones
- se puede utilizar la definición de listas por comprensión

La sintaxis de HASKELL provee ventajas para la evaluación perezosa, se incluyen reglas para realizar cálculos dependiendo de la definición de función utilizada ya sea que se aplique un emparejamiento de patrones, revisión de guardias o uso de definiciones locales (`where`) que ayudan a que los argumentos se evalúen una sóla vez.
La definición de tipos de datos algebraicos ofrece una forma de incluir nuevos tipos de datos mediante el constructor básico `data`:

$$\textbf{data}\ D\ x_1 \ldots x_n = K_1\ t_{11} \ldots t_{1m_1} |\ \ldots\ | K_n\ t_{n1} \ldots t_{nm_n}$$

Donde, D es el tipo de datos que se define y cada K_i es un constructor correspondiente a un valor del tipo de datos. El constructor `data` permite definir tipos recursivos, abstracciones de tipos, productos y sumas[2]. Los operadores y el orden de evaluación de funciones también son importantes para HASKELL, el orden en que las aplicaciones de funciones son evaluadas es de gran ayuda, se puede realizar de afuera hacia adentro o de izquierda a derecha, asimismo todas estas estrategias de evaluación están en correspondencia con el cálculo lambda, base de cualquier lenguaje funcional. Respecto a la persistencia de las estructuras de datos, HASKELL utiliza la recolección de basura para evitar que la memoria se sature al tener estructuras repetidas aunque diferentes. A pesar de que HASKELL es un lenguaje puramente funcional y rígido en cuanto al tipado de funciones, tiene la ventaja de incluir efectos tales como escritura y lectura de archivos o simular la programación secuencial. Estos efectos se manejan mediante el uso de mónadas como se explica en la siguiente sección.

[2] categóricamente

1.3. Mónadas

Como vimos en la sección anterior, el cálculo lambda es considerado como una base fundamental para los lenguajes de programación, pero se debe especificar una semántica que permita razonar respecto a cómputos como son conocidos en ciencias de la computación. Esta semántica la expone E. Moggi en [21], mediante un sistema formal llamado el cálculo lambda computacional[3] ($\lambda_c - calculus$) desarrollado para demostrar la equivalencia de programas. La idea principal de Moggi es tomar la teoría de categorías como una teoría general sobre funciones y desarrollar una semántica para cómputos basado en mónadas, para esto es necesario que los programas sean considerados como objetos de una categoría, en particular se debe formar una categoría de Kleisli para obtener una mónada. Se puede revisar [32] para conceptos básicos y sobre todo para la definición y propiedades de las mónadas, aquí sólo se estudiará a las mónadas dentro del contexto de programación funcional.

El cálculo lambda computacional es un metalenguaje para semánticas denotacionales de los lenguajes de programación, modela fielmente diversas características de un lenguaje de programación que no provee el cálculo lambda tipado ordinario como son el no determinismo, la no terminación, especificar alguna estrategia de evaluación, manejar efectos laterales en una evaluación, etc. Estas características, Moggi las denomina como nociones de computación que se conocen por efectos computacionales.

La correspondencia que muestra Lambek entre la categorías cartesianas cerradas y el cálculo lambda tipado, como se describe en [30], permite a Moggi modelar lo anterior por medio de una diferencia entre valores y cómputos:

Si A se refiere a un valor de un tipo particular entonces
$T(A)$ se refiere a los cómputos de tipo A

La operación $T(\cdot)$ tiene la estructura de una mónada fuerte en una categoría cartesiana cerrada con valores. La idea de la semántica de programas utilizando la teoría de categorías es que un programa denota un morfismo del objeto A, considerado un valor de tipo A, al objeto TB considerado un cómputo de tipo B.

La correspondencia de Curry-Howard extendida a la descrita por Lambek (Curry-Howard-Lambek) permite pasar de la lógica proposicional intuicionista al cálculo lambda tipado y a las categorías cartesianas cerradas. Así el cálculo lambda computacional es una teoría que expresa las nociones de computación como mónadas y corresponde a las categorías cartesianamente cerradas con mónadas fuertes, en la misma forma el cálculo lambda corresponde a las categorías cartesianamente cerradas con productos.

Una mónada es más fácil de entender mediante las ternas de Kleisli:

[3] *The computational lambda calculus*

Definición 1.1 (Terna de Kleisli)**.** *Una terna de Kleisli sobre una categoría C es una terna $(T, \eta, -^*)$, donde $T : Obj(C) \to Obj(C)$, $\eta_A : A \to TA$, $f^* : TA \to TB$ para cada $f : A \to TB$ y se cumplen las siguientes ecuaciones:*

- $\eta_A^* = Id_{TA}$
- $\eta_A; f^* = f$
- $f^*; g^* = (f; g^*)^*$

Intuitivamente se tiene:

- η_A es la inclusión de valores en cómputos
- f^* es la extensión de la función f, que es una función de valores a cómputos, en una función de cómputos a cómputos

Las tres ecuaciones anteriores son mejor conocidas como leyes de las mónadas, las cuales indican que los programas conforman una categoría, la **Categoría de Kleisli** donde:

- el conjunto de morfismos de A a B es $C(A, TB)$
- la identidad en A es η_A
- la composición de morfismos, f seguido de g, es $f; g^*$

El uso de mónadas en programación funcional ha permitido agregar efectos al lenguaje sin perder la propiedad de transparencia referencial. Las mónadas en HASKELL se definen como una familia de tipos `m a` donde `m` es un constructor polimórfico y tiene las funciones return, >>= llamado bind correspondientes a η y $()^\star$ respectivamente además de >> y fail. Asimismo se cumplen las tres leyes de las mónadas:

- `m>>=return  == m`
- `(return x)>>=f == f x`
- `(m>>=f)>>=g == m>>=(λx.f x >>= g)`

Cualquier mónada define una serie de operaciones primitivas basadas en return y >>= agrupadas en una biblioteca particular permitiendo que el usuario las utilice. Mediante la notación `do` es posible simular la programación secuencial combinando operaciones de las bibliotecas incluidas. Uno de los ejemplos iniciales para comprender mónadas es la mónada de entrada/salida:

- el constructor para la mónada es `IO`
- un valor de tipo `IO a` es un cómputo, cuando se ejecuta hace alguna operación de entrada o salida antes de devolver un valor de tipo `a`.

- la mónada `IO` provee una serie de operaciones de entrada y salida como `getLine :: IO String` que espera una línea de entrada y `putStrLn :: String ->IO ()` que despliega una cadena seguida de un salto de línea.

Así se puede tener el siguiente programa:

```
main = do putStrLn "Escribe una linea de texto:"
          x <- getLine
          putStrLn (reverse x)
```

donde `reverse` obtiene la reversa de una cadena o lista.

1.4. Programación politípica

En programación hay muchas veces en que se repiten o reescriben definiciones de funciones ya que son necesarias debido a los diferentes tipos de datos que manejan. Hay muchos ejemplos de esta situación como funciones de igualdad, unificadores, impresiones con formato[4], depuradores[5], etc. Esto puede evitarse mediante el uso de una función "genérica" que sirva para diferentes tipos de datos o estructuras. La programación politípica[6] es un paradigma que engloba algoritmos que tienen que ser implementados una y otra vez para diferentes estructuras o tipos de datos. Una función politípica (*polytipic function*) es una función que que se define por inducción en la estructura de los tipos de datos usados, el politipismo es una herramienta que provee al programador la habilidad de definir funciones de esta forma como se describe en [15, 13].

Las funciones politípicas son generales y abstractas y resultan útiles al desarrollar sistemas computacionales complejos (software) ya que se adaptan automáticamente a los cambios en la estructura. Por ejemplo la función que calcula el tamaño de alguna estructura conserva la misma idea, se debe obtener como resultado el número de partes. Para el tipo de dato de listas, la función `length` recibe una lista y devuelve el número de elementos; para el tipo de árboles n-arios esta función devuelve el número de nodos, así que sólo es necesario conocer el constructor de tipo que recibe como argumento para aplicar la definición.

Hay varias formas de llevar a cabo la programación politípica:

- Utilizar un tipo de dato universal
  Definir las funciones necesarias para el tipo de dato universal así como funciones de traslación entre el tipo universal y los diferentes tipos de dato. Un problema recurrente se presenta en las traslaciones entre los tipos de dato y el tipo universal ya que se puede perder información al pasar de uno a otro.

[4] *pretty printers*
[5] *debuggers*
[6] Polytipic Programming

- Utilizar polimorfismo de orden superior y clases de constructores
  Al utilizar polimorfismo se conserva la información de los tipos pero se crean otros problemas como trabajar con tipos de datos mutuamente recursivos.

- Utilizar una construcción sintáctica especial
  Un ejemplo de este camino es la extensión de Haskell, Polyp [15], incluye una construcción sintáctica para definir funciones mediante clases de Haskell.

Las funciones politípicas son útiles en casos donde los programas son independientes, por ejemplo en sistemas de reescritura como se discute en [15], implementando las funciones politípicas en el sistema Polyp. La característica importante es que las funciones politípicas se adaptan a estructuras que pueden cambiar.

Nosotros adaptamos la segunda alternativa, la cual se utiliza en [13] para implementar la memoización. A continuación se introducen algunos términos necesarios para entender este enfoque en el siguiente capítulo.

1.4.1. *Especies*

Una función politípica se define mediante inducción sobre la estructura de los tipos, para esto es necesario el uso de ***especies***[7], de manera que se pueda considerar a los tipos como valores, formalmente se requiere polimorfismo paramétrico de orden superior. Consideremos la siguiente gramática para construir especies:

$$K ::= \star \mid (K \to K)$$

A partir de un conjunto de variables de tipo X y los constructores $C = \{1, Int, (+), (\times)\}$ podemos generar expresiones de tipo como son comúnmente conocidos en el ámbito de la *teoría de tipos*, a través de la siguiente gramática:

$$T ::= X \mid B \mid T + T \mid T \times T \mid (\Lambda X.K \to T) \mid (TT) \mid (\mu T)$$

Donde X es un conjunto de variables de tipo, B es el conjunto de tipos básicos $\{1, Int\}$, $\Lambda X.K \to T$ denota la abstracción lambda para tipos, TT denota la aplicación de tipos y μT es el mínimo punto fijo de $\Lambda X.K \to T$ útil para definir tipos inductivos. Así, los constructores o primitivas tienen las siguientes especies:

$$\begin{array}{ll} 1, Int & :: \star \\ (+), (\times) & :: \star \to \star \to \star \end{array}$$

Las especies permiten excluir términos que no corresponden a tipos bien especificados[8]. Esta representación de los tipos de datos está basada en teoría de categorías, los tipos de datos se

[7] *Kinds* en inglés.
[8] *well-kinded*

pueden representar mediante objetos iniciales de la categoría de las $F-álgebras$ en donde el funtor F describe la estructura del tipo. En la gramática para tipos, un tipo T corresponde a un funtor F.
Al mantener la descripción del tipo de dato en el funtor se puede facilitar la programación politípica ya que se tienen familias de funciones que se definen por inducción sobre los funtores como se ve en la gramática. En conclusión una función politípica, como se dijo antes, es una función que está definida inductivamente sobre la estructura de los tipos de datos o bien por inducción sobre los funtores.

1.4.2. *Tries* y *Tries* Generalizados

Los árboles de búsqueda digitales o *tries* son una estructura de datos para almacenar información, toman ventaja de la estructura del tipo de dato de las llaves. Son parecidos a los árboles binarios de búsqueda pero pueden tener más descendientes y emplea las llaves de búsqueda como estructuras para organizar la información. La introducción de esta estructura es atribuida a A. Thue en 1912 para representar cadenas, en 1959 se sugiere como una estructura para búsquedas. La generalización de los tries a cadenas de elementos de un tipo de dato fue descubierto en 1979 por C.P. Wadsworth y más tarde, en 1995 se formalizó el concepto bajo la Teoría de Categorías mostrando que un trie es un funtor y la función de búsqueda corresponde a una transformación natural. Los tries tienen ventajas sobre otras estructuras de almacenamiento:

- Ventajas sobre árboles binarios de búsqueda.
  El tiempo de búsqueda en un trie es proporcional al tamaño de la estructura que está almacenada, es decir si la longitud de la estructura es m el tiempo de búsqueda, en el peor de los casos, es de $O(m)$. En un árbol de búsqueda de altura $log(n)$, la búsqueda de la estructura de longitud m realiza en el peor de los casos $O(mlog(n))$ operaciones, ya que se realizan comparaciones de estructuras de esa longitud.
  Además, el espacio de almacenamiento de un trie es menor ya que se comparten prefijos entre los nodos.

- Ventajas sobre tablas dispersas o tablas hash.
  En un trie no se debe dar una función para la búsqueda como es el caso de la función de dispersión o direccionamiento, ni modificarla cuando se agregan más llaves junto con sus valores. Además, en los tries no hay colisiones de llaves como sucede en las tablas dispersas imperfectas en donde se asocia una llave a diferentes valores. Los tries proveen un orden léxico para las llaves.

Una forma de ver a los tries es considerar las aristas que salen de un nodo y verlas como un mapeo que va de un nodo a diferentes tries. Por ejemplo para cadenas[9], a partir del tipo de dato para ellas:

```
type  String  =  [Char]
```

[9]recordemos que en HASKELL una cadena es una lista de caracteres

se puede definir el trie que tiene como llaves cadenas:

```
data MapStr a = TrieStr (Maybe a) (MapChar (MapStr a))
```

donde el mapeo a caracteres está bien definido y el tipo `Maybe` está definido por:
`data Maybe a = Nothing | Just a`
Ahora se puede definir la función de búsqueda

$$\begin{array}{lcl} lookupStr & :: & Str \to MapStr\ v \to v \\ lookupStr\ Nil\ (TrieStr\ Nothing\ tc) & = & \mathsf{error} \\ lookupStr\ Nil\ (TrieStr\ v\ tc) & = & v \\ lookupStr\ (Cons\ c\ s)(TrieStr\ tn\ tc) & = & (lookupStr\ s\ \circ\ lookupChar\ c)\ \ tc \end{array}$$

En [25], C. Okasaki da una implementación de los tries generalizados representándolos por medio de funtores. En [12], R. Hinze define los tries generalizados y sus operaciones para tipos de datos polimórficos de primer orden basándose en la programación politípica.

La generalización de los tries es necesaria para que las llaves no sean sólo cadenas, sino estructuras más complejas como un árbol. Modificar las aristas, para que reflejen la estructura de dato usada como llave, se realiza al extender los mapeos de las aristas al mapeo correspondiente de la nueva estructura. Las llaves pueden ser estructuras que involucren tipos recursivos. Así los tries generalizados son un tipo de dato indexado por tipos.
Con ayuda de la programación politípica se puede definir a los tries generalizados y sus funciones, considérese el tipo de dato que asigna un constructor de tipo a la especie $\star \to \star$ por cada constructor de tipo de la especie $\star$:

$$Map :: \star \to \star \to \star$$

A partir de este tipo es posible implementar todas las operaciones sobre tries generalizados:

$$\begin{array}{lcl} empty\langle k\rangle & :: & \forall v.Map\langle k\rangle\ v \\ single\langle k\rangle & :: & \forall v.k \times v \to Map\langle k\rangle\ v \\ lookup\langle k\rangle & :: & \forall v.k \to Map\langle k\rangle\ v \to Maybe\ v \\ insert\langle k\rangle & :: & \forall v.(v \to v \to v) \to k \times v \to (Map\langle k\rangle\ v \to Map\langle k\rangle\ v) \\ merge\langle k\rangle & :: & \forall v.(v \to v \to v) \to (Map\langle k\rangle\ v \to Map\langle k\rangle\ v \to Map\langle k\rangle\ v) \end{array}$$

Los tries generalizados están basados en los isomorfismos conocidos como leyes de los exponentes:

$$\begin{array}{lcl} 1 \to v & \cong & Maybe\ v \\ (k_1 + k_2) \to v & \cong & (k_1 \to v) \times (k_2 \to v) \\ (k_1 \times k_2) \to v & \cong & k_1 \to (k_2 \to v) \end{array}$$

Donde los mapeos a v son finitos. Estos mapeos pueden ser representados mediante $Map\langle k\rangle v$, así los isomorfismos anteriores se reescriben como:

$$\begin{array}{lcl} Map\langle 1\rangle & = & Maybe \\ Map\langle Int\rangle & = & Patricia.Dict \\ Map\langle k_1 + k_2\rangle & = & Map\langle k_1\rangle \times Map\langle k_2\rangle \\ Map\langle k_1 \times k_2\rangle & = & Map\langle k_1\rangle \cdot Map\langle k_2\rangle \end{array}$$

Donde *Patricia* es una variación de los tries[10], se puede revisar [24] para detalles de la implementación de esta variación. Observemos que para el tipo básico Int se elige una definición particular adecuada. Veamos ahora ejemplos de tries vacíos, unitarios y la función de búsqueda. En estos ejemplos se aprecia la correspondencia de los tipos, es decir los isomorfismos aplicados a los diferentes tipos.

$$\begin{array}{lcl} empty\langle k\rangle & :: & \forall v.Map\langle k\rangle\ v \\ empty\langle 1\rangle & = & Nothing \\ empty\langle Int\rangle & = & Patricia.empty \\ empty\langle k_1 + k_2\rangle & = & (empty\langle k_1\rangle, empty\langle k_2\rangle) \\ empty\langle k_1 \times k_2\rangle & = & empty\langle k_1\rangle \end{array}$$

$$\begin{array}{lcl} single\langle k\rangle & :: & \forall v.k \times v \to Map\langle k\rangle\ v \\ single\langle 1\rangle((), v) & = & Just\ v \\ single\langle Int\rangle(i, v) & = & Patricia.single(i, v) \\ single\langle k_1 + k_2\rangle(Inl\ i_1, v) & = & (single\langle k_1\rangle(i_1, v), empty\langle k_2\rangle) \\ single\langle k_1 + k_2\rangle(Inr\ i_2, v) & = & (empty\langle k_1\rangle, single\langle k_2\rangle(i_2, v)) \\ single\langle k_1 \times k_2\rangle((i_1, i_2), v) & = & single\langle k_1\rangle(i_1, single\langle k_2\rangle(i_2, v)) \end{array}$$

$$\begin{array}{lcl} lookup\langle k\rangle & :: & \forall v.k \to Map\langle k\rangle\ v \to Maybe\ v \\ lookup\langle 1\rangle()\ t & = & t \\ lookup\langle Int\rangle i\ t & = & Patricia.lookup\ i\ t \\ lookup\langle k_1 + k_2\rangle(Inl\ i_1)(t_1, t_2) & = & lookup\langle k_1\rangle i_1\ t_1 \\ lookup\langle k_1 + k_2\rangle(Inr\ i_2)(t_1, t_2) & = & lookup\langle k_2\rangle i_2\ t_2 \\ lookup\langle k_1 \times k_2\rangle(i_1, i_2)\ t_1 & = & \mathbf{do}\{t_2 \leftarrow lookup\langle k_1\rangle i_1\ t_1; lookup\langle k_2\rangle i_2\ t_2\} \end{array}$$

En el siguiente capítulo se describirán dos enfoques de la memoización basados en mónadas y programación politípica para llegar al último capítulo y concentrarse en la memoi zación selectiva.

[10]Patricia trie (tree) o radix tree

Capítulo 2

Memoización

La optimización de programas es un proceso que modifica la estructura del programa para mejorarlo. Existen varias técnicas para la optimización de programas ya sea para llevar a cabo la ejecución en menor tiempo, para utilizar menos recursos computacionales, etc. Hay diferentes niveles de optimización dependiendo del resultado requerido, la memoización, del inglés *memoization*, es una técnica de optimización de programas a nivel de ejecución que evita recalcular resultados durante el cómputo. Esta técnica fue introducida originalmente por Michie [17] en 1968 para reemplazar funciones no lineales por funciones memoizadas.

Una función memoizada (*memo function*) es como una función ordinaria, excepto que recuerda algunos o todos los argumentos que se le han aplicado junto con los resultados calculados. Estos pares, argumento-resultado, se almacenan en una tabla llamada tabla de memoización (*memo table*). Así, cuando una función memoizada es evaluada reutiliza, si es posible, los resultados antes calculados. Esto es, cuando se evalúa con un argumento el cual ya ha sido aplicado anteriormente, se reusa el resultado almacenado en la tabla. Por tanto, la memoización intercambia cómputos que pueden llegar a ser caros por una revisión de la tabla; en términos de complejidad computacional tiempo/espacio, la memoización baja el costo del tiempo de ejecución del programa a cambio de subir el costo de almacenamiento o espacio.
Cualquier función puede ser memoizada pero puede suceder que no mejore su desempeño a pesar de que ésta sea una técnica de optimización. Por ejemplo la función factorial, al tener complejidad lineal ($O(n)$), no aprovecha completamente el reuso de resultados previos ya que sólo utiliza los argumentos una sola vez. Al calcular $fact(n)$ se almacenarán todos los valores, una llamada posterior $fact(2n)$ no reutilizará todos los valores almacenados. Por otro lado, las funciones recursivas pueden tener mejor desempeño al ser memoizadas ya que se reutilizan los resultados de evaluaciones anteriores.

Cada vez que se evalúa una función memoizada se le asocia una tabla para almacenar los argumentos y los resultados obtenidos. La memoización se puede implementar en un sentido funcional puro, como veremos en este capítulo. Sólo que hay una observación importante acerca de la tabla de memoización: esta tabla requiere ser modificada y actualizada durante la evaluación de la función memoizada. Se deben almacenar los pares argumento-resultado

que se calculan en las llamadas recursivas y así reutilizarlos. Las actualizaciones dinámicas de esta tabla pueden ser vistas como operaciones no funcionales debido a que no se pueden modificar estructuras existentes sino crear nuevas. Por tanto, el proceso de memoización requiere que cada función memoizada o funciones que interactúan con funciones memoizadas deban ser modificadas para tratar a la tabla de memoización como parte de sus argumentos y resultados. Utilizar una implementación de la memoización puramente funcional requiere que la tabla de memoización sea un argumento extra de la función y una nueva tabla sea devuelta como resultado. Así una implementación funcional permite que la tabla de memoización sea parte de los argumentos de la función de forma que las tablas se transforman funcionalmente en vez de realizar operaciones destructivas sobre ellas.

Debido a que en la programación funcional es posible construir funciones de orden superior se puede construir la función `memoize`, la cual a partir de una función obtenga una versión memoizada de ella. La función a memoizar debe ser transparente referencialmente ya que una llamada a ella debe ser equivalente a devolver solamente el resultado correspondiente al argumento con el que fue llamada.

Hay tres operaciones auxiliares que deben ser implementadas para manejar la técnica de memoización, en particular sirven para operar la tabla de memoización:

- `in_memo_table` revisa si un argumento ya ha sido utilizado antes y está guardado en la tabla junto con el resultado de su aplicación.
- `lookup` busca el argumento y su valor almacenado.
- `insert` actualiza la tabla al agregar un argumento y el resultado calculado.

Estas operaciones pueden variar en nombre pero mantienen la misma funcionalidad, además pueden reducirse sólo a las últimas dos.
A continuación se desarrolla un ejemplo característico de la programación dinámica, el problema de la subsecuencia común más larga, para exponer cómo se aplica esta técnica a funciones recursivas.

Ejemplo *Dadas dos cadenas o palabras se debe encontrar la longitud de la subsecuencia común más larga sin importar que los símbolos no sean adyacentes.*
Por ejemplo para las cadenas: `[2,4,6,5,3,4,7,9]` `[2,5,4,7,8,9]` se tiene como resultado 5 que es la longitud correspondiente a la subsecuencia `[2,5,4,7,9]`.

Una solución recursiva a este problema es la siguiente:

```
lcs :: [Int] -> [Int] -> Int

lcs xs [] = 0
lcs [] ys = 0
lcs (x:xs) (y:ys)
     |x==y      = 1 + lcs xs ys
     |otherwise = max(lcs (x:xs) ys) (lcs xs (y:ys))
```

El caso en que las cabezas de las listas no sean iguales se deben calcular dos subsoluciones que no son ajenas y que generan a su vez más subsoluciones dependientes de las anteriores. Estas llamadas recursivas generan recálculos que se podrían evitar usando memoización.

Utilizando programación funcional pura se puede mejorar la solución recursiva e incluir la memoización. Como se muestra en [31], un paso intermedio es en tomar las subpartes de las listas y revisarlas.

```
lcs1 :: [Int] -> [Int] -> Int -> Int -> Int

lcs1 xs ys 0 j = 0
lcs1 xs ys i 0 = 0
lcs1 xs ys i j
    | xs!!(i-1) == ys!!(j-1) = (lcs1 xs ys (i-1) (j-1)) + 1
    | otherwise          = max (lcs1 xs ys i (j-1)) (lcs1 xs ys (i-1) j)
```

Esta solución intermedia requiere como argumentos extra las longitudes de las listas a evaluar para obtener un mejor resultado. Finalmente, se agrega la tabla de memoización para almacenar los resultados que se calculan durante la evaluación, esta definición no es inmediata pero es una simulación de la anterior.

```
lcst :: [Int] -> [Int] -> [[Int]]

lcst xs ys = result
    where result = [0,0..]:zipWith f [0..] result
          f i prev = ans
              where ans = 0:zipWith g [0..] ans
                    g j v
                        | xs!!i == ys!!j = prev!!j + 1
                        | otherwise = max v (prev!!(j + 1))
```

Esta definición genera una lista infinta de listas que sirve como tabla, dado que no se conoce el número máximo de pares argumento-resultado que pueden suceder durante la evaluación. Se pueden apreciar las bondades de la evaluación perezosa de Haskell al tener una lista infinita para representar una tabla de memoización. Si se toma la n-ésima lista y su m-ésimo elemento se obtiene el resultado de la subsecuencia común más larga entre las listas que tienen como longitud n y m respectivamente: `lcs1 xs ys n m = (lcst xs ys)!!n!!m`.

Realizando diferentes pruebas con estas definiciones en Haskell se pudo observar que a pesar de que la memoización es una técnica de optimización, el número de reducciones que se realizan con la última definición es mucho mayor que el número de

reducciones en la solución recursiva inmediata para resolver este problema. Por tanto el desempeño de esta implementación de la memoización es deficiente, ya que el número de reducciones varia por mucho en la definición memoizada del problema en comparación con la definición recursiva:

```
Main> lcs [3,2,6,7,9,1] [3,6,7,9]
4
(233 reductions, 303 cells)
Main> lcs1 [3,2,6,7,9,1] [3,6,7,9] 6 4
4
(6081 reductions, 7807 cells)
Main> (lcst [3,2,6,7,9,1] [3,6,7,9])!!6!!4
4
(3309 reductions, 4423 cells)


Main> lcs [1,2,3,4] [4,3,2,1]
1
(418 reductions, 578 cells)
Main> lcs1 [1,2,3,4] [4,3,2,1] 4 4
1
(3342 reductions, 4263 cells)
Main> (lcst [1,2,3,4] [4,3,2,1])!!4!!4
1
(2788 reductions, 3714 cells)
```

Veamos otros enfoques de la memoización que aseguran mejorar el desempeño mediante fundamentos categóricos y programación de orden superior.

2.1. Memoización monádica

Como se discutió en el capítulo anterior, E. Moggi encontró una forma para simplificar el razonamiento de programas que contienen efectos utilizando mónadas. Wadler toma esta idea y propone utilizar a las mónadas dentro de la programación funcional. A partir de esta idea Frost, en [8], describe un proceso sistemático para incorporar mónadas a programas o partes de programas que requieran memoizarse.

A continuación se estudia y modifica el ejemplo de Fibonacci: se utilizan dos mónadas para modificarlo, la primera sólo para reescribirlo en lenguaje monádico y la segunda para incluir un estado en este caso un contador. Los códigos que se muestran a continuación están

traducidos de [8] a sintaxis de HASKELL para poder probarlos, ya que este autor utiliza el lenguaje funcional Miranda[1] para ejemplificar las definiciones.

Recordemos la función de Fibonacci:

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Veamos dos formas de transformar Fibonacci:

- Usando la mónada identidad
  La mónada de identidad `idm A == A` se define mediante las dos funciones de las mónadas:

  ```
  return_id  :: A→ idm A
  return_id  x = x

  bind_id :: idm A → (A → idm B) → idm B
  x bind_id f = f x
  ```

  Por lo tanto Fibonacci se transforma en:

  ```
  fib1 0 = return_id 0
  fib1 1 = return_id 1
  fib1 n = fib1 (n-1) bind_id f
            where f a = fib1 (n-2) bind_id g
                        where g b = return_id (a+b)
  ```

- Usando la mónada de estado
  La mónada de estado `ste A == edo→(A,edo)` donde el estado será un contador `edo == Int`:

  ```
  return_st  :: A→ ste A
  return_st  a = f   where f t = (a,t)

  bind_st  :: ste A → (A → ste B) → ste B
  m bind_st  k = f
              where f x = (b,z)
                          where (b,z) = k a y
                                        where (a,y) = m x
  ```

  Por lo tanto Fibonacci se transforma en:

[1] Miranda es una marca registrada de Research Software Ltd.

```
fib2 0 = return_st 0
fib2 1 = return_st 1


fib2 n = cfib (n-1) bind_st f
         where f a = cfib (n-2) bind_st g
                     where g b = return_st (a+b)

cfib = count fib

count f n c = (res, k+1) where (res, k) = f n c
```

Ahora que se ha ejemplificado la forma de transformar un programa para que incluya mónadas se tomará el segundo ejemplo para incluir tablas de memoización como estado, sólo se modificará la representación del estado. La tabla de memoización está representada como una lista de pares argumento-resultado. El estado es una lista de tablas de memoización, se tiene un par que contiene una cadena para indicar el nombre de la función memoizada y la lista de pares argumento-resultado.

```
edo == [([Char],[(Int,Int)])]

fib 0 = return_st 0
fib 1 = return_st 1
fib n = mfib (n-1) bind_st f
        where f a = mfib (n-2) bind_st g
                    where g b = return_st (a+b)

mfib = memoize ``fib'' fib
```

La función `memoize` es la encargada de manejar la técnica de memoización, define cómo manejar la función que incluye una mónada y las funciones de búsqueda (`lookup`) y de actualización de la tabla (`update`).

```
memoize name f in table
    | table_res == [] = (res, update newtable name inp res)
    | otherwise = (table_res!!0, table)
    where table_res = lookup name inp table
          (res, newtable) = f inp table

lookup name inp table
    | res_in_table == [] = []
    | otherwise = [res | (inp, res) <- (res_in_table!!0)]
    where res_in_table = [pairs | (name, pairs) <- table]
```

```
update [] name inp res = [(name,[(inp, res)])]
update ((key, pairs):rest) name inp res
    | key == name = (key,(inp,res):pairs):rest
    | otherwise = (key,pairs): update rest name inp res
```

En [4], Brown y Cook describen una forma para modificar una función que incluya mónadas, *monadification*. Este proceso es parecido al realizado por Frost sólo que estos autores utilizan HASKELL para mezclar ventajas de dos diferentes paradigmas de programación que mejoren la técnica de memoización: hacen uso de la evaluación perezosa que provee HASKELL junto con el concepto de herencia, como es conocido y usado en la programación orientada a objetos, utilizando clases derivadas, en particular utilizan la subclase *Mixin* para especializar el comportamiento de una variedad de clases, esta subclase permite incluir funcionalidad para ser heredada por otra clase sin tener que definir métodos específicos, se puede decir que es una forma de tener funciones genéricas. Ambas implementaciones sacan ventaja de las mónadas para incluir efectos, la diferencia entre la implementación descrita por Frost y la de Brown y Cook radica en que la primera es puramente funcional ya que utiliza las mónadas para pasar el estado en cada llamada recursiva, es decir pasa la tabla de memoización en forma de estado para que sea utilizada en cada llamada recursiva. En esta nueva implementación, Brown y Cook hacen uso de la notación **do** característica del uso de mónadas en HASKELL para manejar efectos pero sin hacer uso de la mónada de estado como representación de la tabla. Una primera versión del ejemplo para Fibonacci, llamada recursión monádica, utiliza una mónada genérica como parámetro (`Monad m`), devolviendo cómputos en esta mónada mediante `return`:

```
mFib :: Monad m ⇒ Int → m Int
mFib 0 = return 0
mFib 1 = return 1
mFib (n+2) = do a ← mFib n
                b ← mFib (n+1)
                return (a+b)
```

Utilizando la mónada de identidad se obtiene la función original:

```
fib_m :: Int → Int
fib_m = runIdentity ∘ mFib
```

La etiqueta `runIdentity` sirve para representar los valores de una mónada como cómputos. Utilizando recursión abierta y herencia se puede hacer referencia para que la memoización afecte a las llamadas recursivas. La referencia a sí misma debe ser explícita y por tanto al usar puntos fijos es posible combinar generadores que imiten herencia:

```
gFib :: (Int → Int) → (Int → Int)
gFib self 0 = 0
```

```
gFib self 1 = 1
gFib self (n+2) = self n + self (n+1)

fib_g :: Int → Int
fib_g = fix gFib
```

Al combinar las versiones anteriores, la recursión monádica y la abierta, es posible crear un generador monádico de Fibonacci:

```
gmFib :: Monad m ⇒ Gen ( Int → m Int)
gmFib self 0 = return 0
gmFib self 1 = return 1
gmFib self (n+2) = do a ← self n
                      b ← self (n+1)
                      return (a+b)
fib_gm :: Int → Int
fib_gm = runIdentity ∘ (fix gmFib)
```

Finalmente la versión memoizada de Fibonacci se obtiene al implementar la tabla de memoización, se dan dos funciones para manejar la tabla *check* y *store*. Estas dos funciones constituyen un diccionario `Dict a b m` donde `a` corresponde a las llaves, `b` a los valores y `m` a la mónada de estado:

```
type Dict a b m = (a → m (Maybe b),  --check
                   a → b → m())      --store
```

Una vez definida la tabla de memoización, se puede ocupar *mixin* para memoizar:

```
memo :: Monad m ⇒  Dict a b m → Gen (a → mb)
memo (check, store) super a = do b← check a
                                 case b of
                                   Just b → return b
                                   Nothing → do b ← super a
                                                store a b
                                                return b
```

Al combinar el generador de Fibonacci con la memoización anterior se obtiene la versión memoizada de Fibonacci:

```
type Memoized a b m = Dict a b m → a → m b

memoFib :: Monad m ⇒ Memoized Int Int m
memoFib dict = fix (memo dict ∘ gmFib)
```

Estos dos enfoques de la memoización monádica son difíciles de entender si no se tiene una base fuerte en programación funcional y en el uso de mónadas. Son soluciones particulares, aunque describen un proceso claro y sistemático para memoizar una función recursiva.

2.2. Memoización politípica

Otra forma de abordar la memoización es mediante la programación politípica combinándola con la evaluación perezosa como lo hace Hinze en [13]. Para implementar la tabla de memoización, Hinze utiliza la estructura de datos *trie* descrita en el capítulo anterior. Este acercamiento surge para evitar errores que pueden suceder con implementaciones comunes de la tabla memo como lo son las tablas dispersas o estructuras de almacenamiento basadas en árboles. Estas implementaciones toman al argumento de la función memoizada como llave para indexar la tabla, los errores o problemas surgen cuando los argumentos que se convertirán en llaves son estructuras complejas como un árbol o una lista. Usar los tries como estructura para el manejo de la tabla asegura que el tiempo de ejecución de una función memoizada sea independiente del número de valores memoizados.

Dado que una función memoizada está determinada por la tabla, entonces es una función que recibe su entrada (llave) y devuelve un resultado (valor). Las dos funciones principales para manejar la tabla de memoización son la búsqueda de una llave (*apply*) y la inserción de una nueva llave con su valor (*tabulate*).

$$\begin{array}{lcl} Table\langle k :: \star\rangle & :: & \star \to \star \\ \\ apply\langle k\rangle & :: & \forall v.Table\langle k\rangle\ v \to (k \to v) \\ tabulate\langle k\rangle & :: & \forall v.(k \to v) \to Table\langle k\rangle\ v \end{array}$$

El tipo $Table\langle k\rangle\ v$ es el tipo de las tablas indexadas por llaves del tipo k y que almacena valores de tipo v. Esta forma de implementar las funciones memoizadas se apoya en la evaluación perezosa de la programación funcional pura, ya que puede darse el caso en donde el tipo de las llaves k sea una estructura infinita.
Así, la tabla de memoización se puede definir a partir de las funciones que actúan sobre ella y que son inversas:

$$\begin{array}{lcl} memo\langle k\rangle & :: & \forall v.(k \to v) \to (k \to v) \\ \\ memo\langle k\rangle\varphi & = & apply\langle k\rangle(tabulate\langle k\rangle\ \varphi) \end{array}$$

Las tablas de memoización dependen del constructor del tipo de las llaves, por lo que debe considerarse a los tipos de datos como funtores para aprovechar las definiciones de éstos mediante 1, $+$, $\times$[2]. En [13] se definen las tablas politípicamente como:

$$\begin{array}{lcl} Table\langle 1\rangle\ v & = & v \\ Table\langle k_1 + k_2\rangle\ v & = & Table\langle k_1\rangle\ v \times Table\langle k_2\rangle\ v \\ Table\langle k_1 \times k_2\rangle\ v & = & Table\langle k_1\rangle(Table\langle k_2\rangle\ v) \end{array}$$

Esta definición está en relación con los tries, el trie para el tipo unitario es el funtor identidad, el trie para sumas es el funtor de producto y para los productos es la composición de productos. Estas definiciones satisfacen las propiedades de un funtor desde el punto de vista categórico.

[2]Se puede revisar [9] para comprender cómo se definen los tipos de datos categóricamente.

Para especializar $Table\langle 1\rangle$ se utilizarán funtores para especies de tipos, obsérvese que la especie de $Table\langle k\rangle$ depende de la especie de k, así $TABLE$ es una especie indexada por especies.

$$\begin{array}{lcl} TABLE\langle\star\rangle & = & \star \to \star \\ TABLE\langle K \to L\rangle & = & TABLE\langle K\rangle \to TABLE\langle L\rangle \end{array}$$

$$\begin{array}{lcl} Table\langle k :: K\rangle & :: & TABLE\langle K\rangle \\ Table\langle a\rangle & = & table_a \\ Table\langle t\ u\rangle & = & (Table\langle t\rangle)(Table\langle u\rangle) \\ Table\langle \Lambda a.t\rangle & = & \Lambda table_a.Table\langle t\rangle \\ Table\langle \mu a.t\rangle & = & \mu table_a.Table\langle t\rangle \end{array}$$

Las funciones para la tabla se describen a continuación, recordemos que éstas dependen de los constructores y no de un tipo de dato en particular, así mismo sólo se revisarán para los constructores 1, $+$, $\times$ que son indispensables para los ejemplos posteriores.

- *Apply*
  La función politípica para obtener un valor de la tabla a partir de una llave es la siguiente:

$$\begin{array}{lcl} apply\langle k\rangle & :: & \forall v.Table\langle k\rangle\ v \to (k \to v) \\ apply\langle 1\rangle\ t\ () & = & t \\ apply\langle k_1 + k_2\rangle(t_1, t_2)(Inl\ i_1) & = & apply\langle k_1\rangle t_1\ i_1 \\ apply\langle k_1 + k_2\rangle(t_1, t_2)(Inr\ i_2) & = & apply\langle k_2\rangle t_2\ i_2 \\ apply\langle k_1 \times k_2\rangle t\ (i_1, i_2) & = & apply\langle k_2\rangle(apply\langle k_1\rangle t\ i_1)\ i_2 \end{array}$$

- *Tabulate*
  La función politípica para agregar a la tabla una llave y un valor es la inversa de la anterior, es decir a partir de la llave y el valor regresa una nueva tabla:

$$\begin{array}{lcl} tabulate\langle k\rangle & :: & \forall v.(k \to v) \to Table\langle k\rangle\ v \\ tabulate\langle 1\rangle\ \varphi & = & \varphi() \\ tabulate\langle k_1 + k_2\rangle\varphi & = & (tabulate\langle k_1\rangle(\varphi \cdot Inl), tabulate\langle k_2\rangle(\varphi \cdot Inr)) \\ tabulate\langle k_1 \times k_2\rangle\varphi & = & tabulate\langle k_1\rangle(\lambda i_1.tabulate\langle k_2\rangle(\lambda i_2.\varphi(i_1, i_2))) \end{array}$$

Ejemplos:

- Tabla de memoización para números naturales
  Recordemos que los números naturales se definen intuitivamente como $Nat = 1 + Nat$, en sintaxis de HASKELL tenemos:

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

Por tanto la tabla de memoización que tiene como llaves a números naturales es una lista infinita:

$$Table\langle Nat\rangle v \ = v \times Table\langle Nat\rangle \ v$$

```
data TNat v = NNat v (TNat v)
```

donde `NNat` corresponde al constructor de la tabla.
Las funciones de la tabla de memoización son las siguientes:

$$\begin{array}{lcl} applyNat & :: & \forall v.TNat\ v \to (Nat \to v) \\ applyNat(NNat\ tz\ ts)\ Zero & = & tz \\ applyNat(NNat\ tz\ ts)\ (Succ\ n) & = & applyNat\ ts\ n \\ \\ tabulateNat & :: & \forall v.(Nat \to v) \to TNat\ v \\ tabulateNat\ \varphi & = & NNat(\varphi Zero)(tabulateNat(\varphi \cdot Succ)) \end{array}$$

El ejemplo clásico y sencillo donde las llaves de la tabla son números naturales es el de Fibonacci, éste queda de la siguiente forma bajo esta implementación de la técnica de memoización:

```
fib ::  Nat → Nat
fib Zero = Zero
fib (Succ Zero) = Succ Zero
fib (Succ (Succ n))  =  memo-fib n + memo-fib (Succ n)

memo-fib ::  Nat → Nat
memo-fib =  applyNat(tabulateNat fib)
```

- Tabla de memoización para listas
  Las listas cuyos elementos son de tipo A se definen mediante $List\ A = 1 \times List\ A$, en sintaxis de Haskell:

```
data List a = Nil | Cons a (List a)
```

La tabla de memoización es una tabla de orden superior que toma un trie como a y obtiene un trie para `List a`:

$$Table\langle List\rangle\ tablea\ v \ = v \times tablea(Table\langle List\rangle\ tablea\ v)$$

```
data TList ta v = NList v (ta (TList ta v))
```

El constructor $Table\langle List\rangle$ es también una rosadelfa generalizada[3].
Las funciones de la tabla de memoización son las siguientes:

$$\begin{array}{lcl} applyList & :: & \forall ta\ a.(\forall ta\ v \to (a \to v)) \\ applyList\ applya(NList\ tn\ tc)\ Nil & = & tn \\ applyList\ applya(NList\ tn\ tc)\ (Cons\ a\ as) & = & applyList\ applya(applyatc\ a)as \end{array}$$

donde se toma la función *applya* como base ya que *List* es un tipo paramétrico. El tipo correspondiente de *applyList* es uno de rango 2 que puede ser soportado por intérpretes de Haskell. La función *tabulate* se define a continuación, de la misma forma es necesaria la función *tabulatea* como base:

$$\begin{array}{lcl} tabulateList & :: & \forall ta\ a.(\forall v.(a \to v) \to ta\ v) \\ & & \quad \to (\forall w.(List\ a \to w) \to TList\ ta\ w) \\ tabulateList\ tabulatea\ \varphi & = & NList(\varphi Nil) \\ & & \quad (tabulatea\ (\lambda\ a.tabulateList\ tabulatea \\ & & \quad\quad (\lambda\ as.\varphi(Cons\ a\ as)))) \end{array}$$

Otro ejemplo conocido de programación dinámica es el problema de la multiplicación óptima de matrices, a continuación se desarrolla mediante una transformación de una versión recursiva en una memoizada politípicamente.

Ejemplo *Dada una secuencia de dimensiones de matrices* $[d_0, \ldots, d_n]$ *donde la matriz* M_i *tiene dimensiones* $d_{i-1} \times d_i$*, se requiere determinar el costo menor para multiplicarlas. El costo de multiplicar dos matrices con dimensiones* $i \times j$ *y* $j \times k$ *es* $i * j * k$*.*

La siguiente solución es una que tiene orden exponencial:

```
cost :: List Nat →  Nat
cost d
    | n≤ 1 = 0
    | otherwise = minimum [cost (take (i+1) d)
                           + d!!0*d!!i*d!!n + cost (drop i d)
                               | i <- [1..n-1]]
    where n = length d - 1
```

Donde `minimum` es una función que devuelve el mínimo elemento de una lista.
Observemos que el trie para las llaves que son listas es una rosaldelfa generalizada[4].

Memoizando esta función se obtiene una de menor costo:

[3]Véase [3] para detalles de las rosadelfas
[4]generalized rose tree

```
cost :: List Nat → Nat
cost d
    | n≤ 1 = 0
    | otherwise = minimum [memo-cost (take (i+1) d)
                           + d!!0*d!!i*d!!n + memo-cost (drop i d)
                               | i <- [1..n-1]]
    where n = length d - 1

memo-cost :: List Nat → Nat
memo-cost = (applyList apply Nat) ((tabulateList tabulateNat) cost)
```

La idea de esta solución es tomar el mínimo de las subsoluciones, las cuales se generan al multiplicar un número menor de matrices. En este ejemplo se hace evidente el uso de la mejor subsolución no independiente del problema, se deben considerar las subsoluciones más pequeñas o las más interiores para mezclarlas y encontrar la solución inicial. Estas subsoluciones se deben guardar para compararlas con las demás e inclusive reutilizar los resultados en composiciones de subsoluciones futuras. La tabla guarda parejas entrada-resultado que son parejas cuya primer entrada corresponde a listas de dimensiones de matrices y la segunda entrada a listas de resultados dependientes de las subsoluciones generadas por la lista de dimensiones.

Capítulo 3

Memoización selectiva

En la técnica de memoización, la forma usual de indexar la tabla memo es utilizando los argumentos de entrada, pero esta forma de indexación puede pasar por alto la relación que existe entre los datos de entrada y los resultados. Para que la memoización sea efectiva se deben considerar sólo las dependencias necesarias entre entrada y salida. Por ejemplo, consideremos la función `fun f(x,y,z) = if x >0 then y else z`. La pregunta (`x>0`) sirve como punto de decisión haciendo que `f` dependa de ella y cambie de forma dinámica. La llamada `f(1,2,3)` obtendrá un resultado en donde sólo están involucrados `x` y `y`, el valor de `z` resulta irrelevante. Si observamos con más detalle, la evaluación de la función no depende del valor del primer argumento sólo basta con conocer su signo. Así, si se realiza otra llamada por ejemplo `f(3,2,35)`, el resultado es el mismo que se obtiene al procesar la entrada `(1,2,3)` ya que el segundo argumento es igual y la guardia $x > 0$ conserva el mismo valor no requiriendo de un nuevo cálculo. Para hacer eficientes este tipo de funciones se deben modificar las llaves que se usan para indexar las tablas de memoización.

La memoización selectiva presentada en [2] modifica la indexación usual de la tabla, ésta se realiza mediante una lista de eventos que guardan el flujo de control de la información de entrada. Esta lista de eventos o rama (*branch*) indica los puntos de elección de acuerdo al tipo de entrada, estos puntos deben revelar las dependencias entre la entrada y el resultado, condición principal para desarrollar adecuadamente la memoización. A pesar de esta modificación en la técnica se pueden dar casos en donde el recálculo de resultados sea inevitable como por ejemplo en `f(-1,5,2)`, dado que el primer argumento corresponde a un evento diferente: `not` (`x` > 0).
Asimismo, en [2], se propone un sistema en donde se incluye el tipo modal $!\mathsf{T}$ (bang) para un manejo especial de las funciones a memoizar. El tipo bang requiere que el tipo subyacente T sea indexable, es decir que acepte una función inyectiva $i : \mathsf{T} \rightarrow \mathsf{Int}$ llamada función de indexación. Cuando el tipo T no es indexable se deben guardar los valores de ese tipo en "cajas", para ello, en el artículo mencionado se sugiere el uso del tipo $\Box\,\mathsf{T}$. Los tipos indexables y el tipo $\Box\,\mathsf{T}$ para tipos no indexables permiten una implementación del sistema en donde las tablas de memoización están representadas mediante tablas dispersas (*hash tables*), mejorando el desempeño de la biblioteca propuesta y desarrollada.

El tipo $\Box\,\mathsf{T}$ o $\mathsf{box}\,\mathsf{T}$ es usado en sistemas que reflejan el comportamiento del almacenamiento, como por ejemplo el sistema de referencias expuesto en el capítulo 13 de [27]. Las locaciones dentro de una memoria sirven como índice a los valores guardados en ellas ya que son únicas. Por otra parte, comparando la sintaxis del sistema con la que aparece en [26] se puede ver la similitud entre las reglas del tipo bang y las del tipo modal para explicar la necesidad. Pero de la misma manera se puede ver la semejanza entre este tipo modal y el de referencias.

En este capítulo se retoma la memoización selectiva haciendo algunos cambios pero conservando las ideas principales. Nuestras aportaciones incluyen:

- eliminación del uso del tipo modal bang para incluir el tipo box como el tipo que distingue a la memoización dentro del nuevo sistema, es decir que el tipo box se utilizará para la indexación de tipos y para llevar el recuento de las dependencias de dato se usará el constructor let box.

- incorporación de un contexto de etiquetas para hacer uso explícito de la memoria.

- desarrollo de una demostración más clara de la seguridad del nuevo sistema mediante traducciones a sistemas sin efectos, es decir sin memoización.

Conservando los siguientes:

- el uso de tipos modales para mostrar los efectos laterales al realizar un cómputo ya que se considerará como efecto lateral la actualización de la tabla de memoización.

- la distinción entre términos y expresiones siendo estas últimas evaluadas respecto a una tabla de valores previos.

- el uso de expresiones como cuerpo de las funciones memoizadas y el uso del constructor return para identificar el cálculo que genera el resultado a almacenar.

- consideración de los argumentos de las funciones memoizadas como *recursos* para explorar incrementalmente el tipo de los mismos.

- mantener una rama que "registre" las dependencias entre la entrada y el resultado derivadas de la exploración mencionada en el punto anterior.

Las aportaciones son derivadas de una revisión exhaustiva del sistema propuesto por Acar et al., constituyen una reestructuración que busca una reformulación del sistema bajo la correspondencia Curry-Howard. Asimismo y sobre todo se buscó una demostración clara de la seguridad del sistema para la memoización selectiva que resultó en el fin principal de este trabajo.

3.1. Un sistema para la memoización selectiva

En esta sección se presenta el sistema para la memoización selectiva (SM), basado en el original que aparece en [2] pero con algunas diferencias o cambios. Estos cambios están hechos para que el sistema SM sea un sistema de tipos independiente de un lenguaje de programación.
Se eliminan los tipos indexables y los tipos bang incorporando los tipos box, también se agregan funciones sin memoizar, el término case y las proyecciones de un término. La semántica estática está definida usando juicios de tipos, en ellos se agrega un nuevo contexto para mantener las etiquetas de las tablas de memoización. La diferencia entre términos y expresiones es necesaria ya que al evaluarlos un término corresponde a un programa y se evalúa de forma común, mientras que las expresiones se evaluarán respecto a una tabla de memoización.

A continuación se describe la sintaxis la cual incluye el tipo modal $\Box$ para indicar necesidad, estudiado en la sección de preliminares y analizado a fondo en [26]:

- *Tipos*. Los tipos están construidos a partir de un conjunto básico de tipos B que incluye al tipo unitario Unit y a los enteros Int.

$$\mathsf{T} ::= \mathsf{B} \mid \mathsf{T} \to \mathsf{T} \mid \mathsf{T} + \mathsf{T} \mid \mathsf{T} \times \mathsf{T} \mid \Box\,\mathsf{T}$$

- *Términos*. Los términos están construidos a partir de un conjunto infinito de variables de término x, un conjunto infinito de variables de recurso a y una serie de operadores primitivos o incluyendo a los números n y el único objeto $\star$ de tipo Unit

$$\begin{array}{rcl} t, r, s & ::= & x \mid a \mid o(t, \ldots, t) \mid \star \mid n \mid \lambda x : \mathsf{T}\,.r \mid \mathsf{mfun}_\ell(f.a.e) \mid rs \mid \mathsf{inl}_\mathsf{T}\, r \mid \mathsf{inr}_\mathsf{T}\, s \mid \\ & & \mathsf{case}(r, x.s, y.t) \mid \langle r, s\rangle \mid \mathsf{fst}\, r \mid \mathsf{snd}\, r \mid \mathsf{box}\, t \end{array}$$

  El término $\mathsf{box}\, t$ es el constructor de términos tipados mediante box, el término que define una función memoizada que generalmente es recursiva es $\mathsf{mfun}_\ell(f.a.e)$, su nombre es (f) y está etiquetada con la metavariable ℓ que pertenece a un conjunto $\mathcal{L}$ de etiquetas que es disjunto de las variables ordinarias y de los recursos. La notación usada en estos términos es llamada *notación de punto*. Esta notación permite identificar a las variables que se encuentran ligadas en un término o expresión, es decir, en expresiones de la forma $x.t$ las presencias de la variable x en t se consideran ligadas. Este mecanismo elimina el uso de paréntesis, el punto indica que se abre un paréntesis que cierra lo más a la derecha posible, sintácticamente hablando.

  La sintaxis de los términos es la llamada abstracta determinada por la gramática anterior. Los términos en sintaxis abstracta corresponden a los siguientes en sintaxis

concreta:

$suma(r, s)$	$r + s$
$resta(r, s)$	$r - s$
$\mathsf{mfun}_\ell(f.a.e)$	$\mathsf{mfun}_\ell\, f(a :: \mathsf{T}) = e$
$\mathsf{case}(r, x.s, y.t)$	$\mathsf{case}\ r\mathsf{of}\ x \Rightarrow s, y \Rightarrow t)$

Los operadores primitivos corresponden a operaciones de los tipos básicos. La estructura de control $\mathsf{if}\ r\ \mathsf{then}\ s\ \mathsf{else}\ t$ se puede ver como azúcar sintáctica de case: $case\ r\mathsf{of}\ true \Rightarrow s, false \Rightarrow t$.

Tanto la notación de punto como la sintaxis abstracta también se utilizan para las expresiones del sistema que a continuación se definen.

- *Expresiones.* Se construyen a partir de términos:

$$e \ ::= \ \mathsf{return}\ t \mid \mathsf{let\,box}\,(t,\, x.e) \mid \mathsf{letprod}\,(t,\, a_1.a_2.e) \mid \mathsf{mcase}\,(t,\, a_1.e_1,\, a_2.e_2)$$

La sintaxis concreta de las expresiones es la siguiente:

$\mathsf{let\,box}(t,\, x.e)$	$\mathsf{let\,box}\, x = t\ \mathsf{in}\ e$
$\mathsf{letprod}(t,\, a_1.a_2.e)$	$\mathsf{letprod}\langle a_1, a_2\rangle = t\ \mathsf{in}\ e$
$\mathsf{mcase}(t,\, a_1.e_1,\, a_2.e_2)$	$\mathsf{mcase}\, t\ \mathsf{of}\ a_1 \Rightarrow e_1 \mid a_2 \Rightarrow e_2$

El uso de expresiones permite separar las dependencias de dato y las de control utilizando variables ordinarias de tipo modal y variables de recursos respectivamente. El uso de la expresión return no tiene relación alguna con la función de mónadas. Aquí se utiliza para denotar los posibles efectos que hay al evaluar las expresiones respecto a una tabla, esto se entenderá mejor en la definición formal de semántica. Asimismo el constructor return permite considerar cualquier término como una expresión.

- *Contextos.* Los contextos son conjuntos finitos de parejas; hay tres clases, los contextos de variables Γ, los contextos de recursos Δ y contextos de etiquetas Σ:

$$\Gamma \ ::= \ \cdot \mid \Gamma, x : \mathsf{T} \qquad \Delta \ ::= \ \cdot \mid \Delta, a :: \mathsf{T} \qquad \Sigma \ ::= \ \cdot \mid \Sigma, \ell : \mathsf{T}$$

donde $\cdot$ denota al conjunto vacío y en $\Gamma, x : \mathsf{T}$ se entiende que x no aparece en Γ de igual forma para los otros contextos $\Delta,\ \Sigma$.
Haciendo referencia al uso de contextos que aparece en [26], los contextos de variables corresponden a contextos de validez y los contextos de recursos a contextos de verdad. El tercer contexto se agregó para llevar un recuento de las etiquetas que figuran en una expresión. Este contexto permite asegurar estáticamente que las etiquetas de diferentes funciones de un mismo programa sean diferentes.

3.1.1. Sistema de tipos

La semántica estática se presenta como un sistema de tipos que deriva juicios de la forma

$$\Gamma \mid \Delta \mid \Sigma \vdash r : \mathsf{T}$$

Estos juicios denotan que un término o una expresión r está bien tipado bajo los contextos Γ, Δ y Σ. Las reglas para derivar estos juicios se definen como sigue:

$$\frac{}{\Gamma, x : \mathsf{T} \mid \Delta \mid \Sigma \vdash x : \mathsf{T}}\ (\mathsf{Tvar}) \qquad \frac{}{\Gamma \mid \Delta, a :: \mathsf{T} \mid \Sigma \vdash a : \mathsf{T}}\ (\mathsf{Tresource})$$

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash t_i : \mathsf{T}_i \quad (1 \leq i \leq n) \qquad \vdash o : \mathsf{T}_1 \times \cdots \times \mathsf{T}_n \to \mathsf{T}}{\Gamma \mid \Delta \mid \Sigma \vdash o\,(t_1, \ldots, t_n) : \mathsf{T}}\ (\mathsf{Tbasicop})$$

$$\frac{}{\Gamma \mid \Delta \mid \Sigma \vdash \star : \mathsf{Unit}}\ (\mathsf{Tunit}) \qquad \frac{}{\Gamma \mid \Delta \mid \Sigma \vdash n : \mathsf{Int}}\ (\mathsf{Tnum})$$

$$\frac{\Gamma, x : \mathsf{T}_1 \mid \Delta \mid \Sigma \vdash t : \mathsf{T}_2}{\Gamma \mid \Delta \mid \Sigma \vdash \lambda x : \mathsf{T}_1 . t : \mathsf{T}_1 \to \mathsf{T}_2}\ (\mathsf{Tlam})$$

$$\frac{\Gamma, f : \mathsf{T}_1 \to \mathsf{T}_2 \mid \Delta, a :: \mathsf{T}_1 \mid \Sigma \vdash e : \mathsf{T}_2}{\Gamma \mid \Delta \mid \Sigma, \ell : \mathsf{T}_2 \vdash \mathsf{mfun}_\ell\,(f.a.e) : \mathsf{T}_1 \to \mathsf{T}_2}\ (\mathsf{Tmfun})$$

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash t_1 : \mathsf{T}_1 \to \mathsf{T}_2 \qquad \Gamma \mid \Delta \mid \Sigma \vdash t_2 : \mathsf{T}_1}{\Gamma \mid \Delta \mid \Sigma \vdash t_1\, t_2 : \mathsf{T}_2}\ (\mathsf{Tapply})$$

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash t : \mathsf{T}_1}{\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{inl}_{\mathsf{T}_2}\, t : \mathsf{T}_1 + \mathsf{T}_2}\ (\mathsf{Tinl}) \qquad \frac{\Gamma \mid \Delta \mid \Sigma \vdash t : \mathsf{T}_2}{\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{inr}_{\mathsf{T}_1}\, t : \mathsf{T}_1 + \mathsf{T}_2}\ (\mathsf{Tinr})$$

$$\frac{\begin{array}{rl}\Gamma \mid \Delta \mid \Sigma & \vdash\ t : \mathsf{T}_1 + \mathsf{T}_2 \\ \Gamma, x_1 : \mathsf{T}_1 \mid \Delta \mid \Sigma & \vdash\ t_1 : \mathsf{T} \\ \Gamma, x_2 : \mathsf{T}_2 \mid \Delta \mid \Sigma & \vdash\ t_2 : \mathsf{T}\end{array}}{\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{case}\,(t,\, x_1.t_1,\, x_2.t_2) : \mathsf{T}}\ (\mathsf{Tcase})$$

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash t_1 : \mathsf{T}_1 \qquad \Gamma \mid \Delta \mid \Sigma \vdash t_2 : \mathsf{T}_2}{\Gamma \mid \Delta \mid \Sigma \vdash \langle t_1, t_2 \rangle : \mathsf{T}_1 \times \mathsf{T}_2}\ (\mathsf{Tpair})$$

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash t : \mathsf{T}_1 \times \mathsf{T}_2}{\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{fst}\, t : \mathsf{T}_1}\ (\mathsf{Tfst}) \qquad \frac{\Gamma \mid \Delta \mid \Sigma \vdash t : \mathsf{T}_1 \times \mathsf{T}_2}{\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{snd}\, t : \mathsf{T}_2}\ (\mathsf{Tsnd})$$

$$\frac{\Gamma \mid \cdot \mid \Sigma \vdash t : \mathsf{T}}{\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{box}\, t : \Box\, \mathsf{T}}\ (\mathsf{Tbox})$$

La regla (Tbasicop) sirve para tipar un operador primitivo o como las operaciones entre enteros u operaciones de comparación, todas estas operaciones se suponen definidas. Este juicio está generalizado, se puede particularizar de acuerdo a la operación requerida, por

ejemplo:

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash t_1 : \mathsf{Int} \quad \Gamma \mid \Delta \mid \Sigma \vdash t_2 : \mathsf{Int} \qquad \vdash \mathit{suma} \ : \mathsf{Int} \times \mathsf{Int} \to \mathsf{Int}}{\Gamma \mid \Delta \mid \Sigma \vdash \mathit{suma}\ (t_1, t_2) : \mathsf{Int}} \ (\mathsf{Tsuma})$$

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash t_1 : \mathsf{Int} \quad \Gamma \mid \Delta \mid \Sigma \vdash t_2 : \mathsf{Int} \qquad \vdash \mathit{mq} \ : \mathsf{Int} \times \mathsf{Int} \to \mathsf{bool}}{\Gamma \mid \Delta \mid \Sigma \vdash \mathit{mq}\ (t_1, t_2) : \mathsf{bool}} \ (\mathsf{Tmenorque})$$

Asimismo la regla (Tmfun) expande el contexto de etiquetas asegurando que una nueva función memoizada tenga asociada una etiqueta única. Para la regla (Tbox) se requiere que el contexto de recursos sea vacío asegurando que el término t no tenga recursos libres antes de una encapsulación en $\mathsf{box}\, t$, esto es necesario para un comportamiento operacional adecuado como se discute para la regla en lógica modal en [26].

Las expresiones se tipan de la siguiente forma:

$$\frac{\Gamma \mid \cdot \mid \Sigma \vdash t : \mathsf{T}}{\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{return}\ t\ : \mathsf{T}} \ (\mathsf{Treturn})$$

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash t : \Box\, \mathsf{T}_1 \qquad \Gamma, x : \mathsf{T}_1 \mid \Delta \mid \Sigma \vdash e : \mathsf{T}_2}{\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{let\,box}(t,\ x.e) : \mathsf{T}_2} \ (\mathsf{Tletbox})$$

$$\frac{\Gamma \mid \Delta \mid \Sigma \vdash t : \mathsf{T}_1 \times \mathsf{T}_2 \qquad \Gamma \mid \Delta, a_1 :: \mathsf{T}_1,\ a_2 :: \mathsf{T}_2 \mid \Sigma \vdash e : \mathsf{T}}{\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{letprod}\,(t,\ a_1.a_2.e) : \mathsf{T}} \ (\mathsf{Tletprod})$$

$$\frac{\begin{array}{rcl} \Gamma \mid \Delta \mid \Sigma & \vdash & t : \mathsf{T}_1 + \mathsf{T}_2 \\ \Gamma \mid \Delta,\ a_1 :: \mathsf{T}_1 \mid \Sigma & \vdash & e_1 : \mathsf{T} \\ \Gamma \mid \Delta,\ a_2 :: \mathsf{T}_2 \mid \Sigma & \vdash & e_2 : \mathsf{T} \end{array}}{\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{mcase}\,(t,\ a_1.e_1,\ a_2.e_2) : \mathsf{T}} \ (\mathsf{Tmcase})$$

La regla (Treturn) permite incluir a los términos como expresiones, esta regla no tiene una análoga en lógica pero permitirá modelar la memoización selectiva. El término requiere estar libre de recursos para asegurar que ninguna dependencia generada por un argumento de una función memoizada se haga explícita en el código antes de incluir el constructor de expresión return. La regla (Tletbox) sirve como regla de eliminación para los tipos box. El constructor let box liga un valor en una caja a una variable ordinaria, de igual forma esta regla tiene su similar en la lógica como se puede apreciar en [26]. Para ligar recursos a otros términos se utilizan los constructores letprod y mcase. De esta forma la expresión let box crea una dependencia de dato, mcase una dependencia de control y letprod no crea dependencias.

3.1.2. Semántica dinámica

La semántica dinámica está dada por reglas mutuamente definidas de evaluación de paso grande, éstas involucran memorias en donde se almacenan tablas de memoización indexadas por ramas. Para términos se tiene la relación $\mu,\ t \Downarrow^t\ v,\ \mu'$ que es muy semejante a cualquier relación de evaluación de paso grande; interactúa con la semántica para expresiones al usar la regla de aplicación de una función memoizada. Recordemos que las expresiones son evaluadas respecto a una tabla de memoización, así la semántica para expresiones $\mu,\ e \Downarrow^e_{\beta@\ell}\ v,\ \mu'$ realiza la evaluación respecto a la tabla que está almacenada en la memoria μ en la locación ℓ, así mismo está indexada mediante ramas con las cuales se accesa a un valor. A continuación se definen formalmente estos conceptos.

El conjunto de valores es un subconjunto de términos definidos como sigue:

$$v ::= \star \mid n \mid \lambda x : \mathsf{T}\,.t \mid \mathsf{mfun}_\ell(f.a.e) \mid \mathsf{inl}\,v \mid \mathsf{inr}\,v \mid \langle v, v\rangle \mid \mathsf{box}\,v$$

Este conjunto de valores está cuidadosamente definido ya que no contiene ni variables ni recursos en contraste con el conjunto de valores que aparece en [2].

Definición 3.1. *Un evento señala el punto de decisión en la evaluación de una expresión. Tal punto se obtiene en el análisis de casos o por un valor de tipo* box *denotado por* $!v$.

$$\varepsilon\ ::= !v \mid \mathsf{inl} \mid \mathsf{inr}$$

Definición 3.2. *Una rama β es una lista de eventos ε, puede ser la rama vacía o la concatenación de un evento y una rama:*

$$\beta\ ::=\ \bullet \mid \varepsilon \cdot \beta$$

Definición 3.3. *Una tabla de memoización θ es una función parcial $\theta : \mathcal{B} \to \mathcal{V}_\mathsf{T}$ que mapea ramas a valores de cierto tipo T. Escribimos $\theta[\beta \mapsto v]$ para la extensión de θ que liga a β con v, asumiendo siempre que $\beta \notin dom(\theta)$.*

Definición 3.4. *Una memoria μ es una función parcial con dominio finito $\mu : \mathcal{L} \to \mathcal{T}$ que mapea etiquetas de locaciones ℓ a tablas memo θ. Una memoria es* inicial *si y sólo si contiene sólo tablas memo vacías, esto es $\forall \ell \in dom(\mu)(\mu(\ell) = \varnothing)$.*
Escribimos $\mu[\ell \mapsto \theta]$ para la extensión de μ que liga a ℓ con θ, siempre asumiendo que $\ell \notin dom(\mu)$. Más aun, cuando $\ell \in dom(\mu)$, escribimos $\mu[\ell \leftarrow \theta]$ para la actualización de la memoria μ que liga ℓ a θ.

La semántica de evaluación para términos está dada por la relación $\mu, t \Downarrow^t v, \mu'$ que modela el proceso de evaluación del término t respecto a la memoria μ resultando en el valor final v y en la memoria μ', está definida por las siguientes reglas de inferencia correspondientes a la relación[1] descrita en [2]:

[1] $\sigma, t \Downarrow^t v, \sigma'$

$$\frac{\mu,\ t_1 \Downarrow^t v_1,\ \mu_1 \quad \ldots \quad \mu_{n-1},\ t_n \Downarrow^t v_n,\ \mu_n \quad \mu_n,\ o(v_1,\ldots,v_n) \Downarrow^t v, \mu'}{\mu,\ o\ (t_1,\ldots,t_n) \Downarrow^t v,\ \mu'}\ (\mathsf{Ebasicop})$$

$$\frac{}{\mu,\ \star \Downarrow^t\ \star,\ \mu}\ (\mathsf{Eunit}) \qquad \frac{}{\mu,\ n \Downarrow^t\ n,\ \mu}\ (\mathsf{Enum})$$

$$\frac{}{\mu,\ \lambda x : \mathsf{T}\,.t \Downarrow^t\ \lambda x : \mathsf{T}\,.t,\ \mu}\ (\mathsf{Elam})$$

$$\frac{\ell \notin dom(\mu)}{\mu,\ \mathsf{mfun}_\ell\,(f.a.e) \Downarrow^t\ \mathsf{mfun}_\ell\,(f.a.e),\ \mu[\ell \mapsto \varnothing]}\ (\mathsf{Emfun})$$

$$\frac{\ell \in dom(\mu)}{\mu,\ \mathsf{mfun}_\ell\,(f.a.e) \Downarrow^t\ \mathsf{mfun}_\ell(f.a.e),\ \mu}\ (\mathsf{Emfun-in})$$

$$\frac{\begin{array}{rl}\mu,\ t_1 & \Downarrow^t\ \lambda x : \mathsf{T}\,.t,\ \mu_1 \\ \mu_1,\ t_2 & \Downarrow^t\ v',\ \mu_2 \\ \mu_2,\ t[x := v'] & \Downarrow^t\ v,\ \mu'\end{array}}{\mu,\ t_1 t_2 \Downarrow^t\ v,\ \mu'}\ (\mathsf{Eapply})$$

$$\frac{\begin{array}{rl}\mu,\ t_1 & \Downarrow^t\ v_1 = \mathsf{mfun}_\ell\,(f.a.e),\ \mu_1 \\ \mu_1,\ t_2 & \Downarrow^t\ v_2,\ \mu_2 \\ \mu_2,\ e[f, a := v_1, v_2] & \Downarrow^e_{\bullet@\ell}\ v,\ \mu'\end{array}}{\mu,\ t_1 t_2 \Downarrow^t\ v,\ \mu'}\ (\mathsf{Emapply})$$

$$\frac{\mu,\ t \Downarrow^t\ v,\ \mu'}{\mu,\ \mathsf{inl}\,t \Downarrow^t\ \mathsf{inl}\,v,\ \mu'}\ (\mathsf{Einl}) \qquad \frac{\mu,\ t \Downarrow^t\ v,\ \mu'}{\mu,\ \mathsf{inr}\,t \Downarrow^t\ \mathsf{inr}\,v,\ \mu'}\ (\mathsf{Einr})$$

$$\frac{\begin{array}{rl}\mu,\ t & \Downarrow^t\ \mathsf{inl}\,v,\ \mu_1 \\ \mu_1, t_1[x_1 := v] & \Downarrow^t\ v_1,\ \mu'\end{array}}{\mu,\ \mathsf{case}\,(t,\ x_1.t_1,\ x_2.t_2) \Downarrow^t\ v_1,\ \mu'}\ (\mathsf{Ecase-l}) \qquad \frac{\begin{array}{rl}\mu,\ t & \Downarrow^t\ \mathsf{inr}\,v,\ \mu_1 \\ \mu_1, t_2[x_2 := v] & \Downarrow^t\ v_2,\ \mu'\end{array}}{\mu,\ \mathsf{case}\,(t,\ x_1.t_1,\ x_2.t_2) \Downarrow^t\ v_2,\ \mu'}\ (\mathsf{Ecase-r})$$

$$\frac{\begin{array}{rl}\mu,\ t_1 & \Downarrow^t\ v_1,\ \mu_1 \\ \mu_1,\ t_2 & \Downarrow^t\ v_2,\ \mu'\end{array}}{\mu,\ \langle t_1, t_2\rangle \Downarrow^t\ \langle v_1, v_2\rangle,\ \mu'}\ (\mathsf{Epair})$$

$$\frac{\mu,\ t \Downarrow^t\ \langle v_1,\ v_2\rangle,\ \mu'}{\mu,\ \mathsf{fst}\,t \Downarrow^t\ v_1,\ \mu'}\ (\mathsf{Efst}) \qquad \frac{\mu,\ t \Downarrow^t\ \langle v_1,\ v_2\rangle,\ \mu'}{\mu,\ \mathsf{snd}\,t \Downarrow^t\ v_2,\ \mu'}\ (\mathsf{Esnd})$$

$$\frac{\mu,\ t \Downarrow^t\ v,\ \mu'}{\mu,\ \mathsf{box}\,t \Downarrow^t\ \mathsf{box}\,v,\ \mu'}\ (\mathsf{Ebox})$$

Observemos que una función memoizada es un valor, pero su evaluación necesita de la creación de una tabla de memoización nueva y vacía cuando la función no tenga una tabla asignada. También la aplicación de una función memoizada recae en la evaluación de la expresión involucrada iniciando en la rama vacía para así explorar las dependencias generadas por esa entrada.

La semántica para expresiones está dada por la relación $\mu, e \Downarrow^e_{\beta@\ell} v, \mu'$, en donde la evaluación de la expresión e se realiza respecto a la memoria μ, terminando en el valor v y en la memoria μ'; todo esto de acuerdo a la consulta realizada por la rama β en la tabla de memoización almacenada en la locación ℓ en μ.
Las reglas se presentan a continuación[2]:

$$\frac{\mu(\ell)(\beta) = v}{\mu,\ \mathsf{return}\ t \Downarrow^e_{\beta@\ell}\ v,\ \mu}\ (\mathsf{Efound})$$

$$\frac{\begin{array}{c}\mu(\ell) = \theta \\ \beta \notin dom(\theta) \\ \mu,\ t \Downarrow^t\ v,\ \mu' \\ \mu'(\ell) = \theta'\end{array}}{\mu,\ \mathsf{return}\ t \Downarrow^e_{\beta@\ell}\ v,\ \mu'[\ell \leftarrow \theta'[\beta \mapsto v]]}\ (\mathsf{Enotfound})$$

$$\frac{\mu,\ t\ \Downarrow^t\ \mathsf{box}\, v,\ \mu_1 \quad \mu_1,\ e[x := v]\ \Downarrow^e_{!v\cdot\beta@\ell}\ v',\ \mu'}{\mu, \mathsf{let\,box}\,(t,\ x.e) \Downarrow^e_{\beta@\ell}\ v',\ \mu'}\ (\mathsf{Eletbox})$$

$$\frac{\mu,\ t\ \Downarrow^t\ \langle v_1,\ v_2\rangle,\ \mu_1 \quad \mu_1,\ e[a_1, a_2 := v_1,\ v_2]\ \Downarrow^e_{\beta@\ell}\ v,\ \mu'}{\mu,\ \mathsf{letprod}\,(t,\ a_1.a_2.e) \Downarrow^e_{\beta@\ell}\ v,\ \mu'}\ (\mathsf{Eletprod})$$

$$\frac{\mu,\ t\ \Downarrow^t\ \mathsf{inl}\, v,\ \mu_1 \quad \mu_1, e_1[a_1 := v]\ \Downarrow^e_{\mathsf{inl}\cdot\beta@\ell}\ v_1,\ \mu'}{\mu,\ \mathsf{mcase}\,(t,\ a_1.e_1,\ a_2.e_2) \Downarrow^e_{\beta@\ell}, v_1,\ \mu'}\ (\mathsf{Emcase-l})$$

$$\frac{\mu,\ t\ \Downarrow^t\ \mathsf{inr}\, v,\ \mu_1 \quad \mu_1, e_2[a_2 := v]\ \Downarrow^e_{\mathsf{inr}\cdot\beta@\ell}\ v_2,\ \mu'}{\mu,\ \mathsf{mcase}\,(t,\ a_1.e_1,\ a_2.e_2) \Downarrow^e_{\beta@\ell}, v_2,\ \mu'}\ (\mathsf{Emcase-r})$$

En la regla (Enotfound) se debe hacer una observación importante, para que la condición $\theta'[\beta \mapsto v]$ pueda ser definida se debe tener que $\beta \notin dom(\theta')$, pero la regla sólo necesita que se cumpla $\beta \notin dom(\theta)$. La correctud de esta regla se justificará más adelante, ver el lema 3.6.
Para evaluar la expresión let box en la memoria μ respecto a la tabla almacenada bajo ℓ con

[2]Están en correspondencia con las que aparecen en [2] bajo la relación $\sigma, l : \beta, e \Downarrow^e v, \sigma'$

la llave β se requiere evaluar el argumento de $\mathsf{let\,box}$ resultando en un valor de la forma $\mathsf{box}\,v$ para sustituir x por v en el cuerpo de la expresión. El valor del argumento debe registrarse en la rama para llevar el recuento de la dependencia de dato. Finalmente, el resultado v' de la evaluación de la expresión $\mathsf{let\,box}$ es la evaluación del cuerpo e modificado.
Como se mencionó antes, la expresión $\mathsf{letprod}$ no genera dependencias que se registran en la rama pero de igual forma se evalúa el argumento que debe ser un par para sustituirlo en el cuerpo de la expresión que al evaluarse genera el resultado v.
Por otro lado, la expresión mcase genera una dependencia de control que se registra en la rama. La evaluación del argumento debe resultar en un caso (inl o inr) para sustituir el valor interior en el cuerpo de la expresión mcase. La evaluación de este cuerpo modificado es el resultado v'.

Las propiedades de sustitución ocupadas en algunas reglas se demuestran a continuación:

Lema 3.1 (Lema de sustitución)**.**

1. *Si* $\Gamma, x : \mathsf{R}\,|\Delta|\Sigma \vdash t : \mathsf{T}$ *y* $\Gamma|\cdot|\Sigma \vdash r : \mathsf{R}$ *entonces* $\Gamma|\Delta|\Sigma \vdash t[x := r] : \mathsf{T}$.

2. *Si* $\Gamma, x : \mathsf{R}\,|\Delta|\Sigma \vdash e : \mathsf{T}$ *y* $\Gamma|\cdot|\Sigma \vdash r : \mathsf{R}$ *entonces* $\Gamma|\Delta|\Sigma \vdash e[x := r] : \mathsf{T}$.

3. *Si* $\Gamma|\Delta, a :: \mathsf{R}\,|\Sigma \vdash t : \mathsf{T}$ *y* $\Gamma|\Delta|\Sigma \vdash r : \mathsf{R}$ *entonces* $\Gamma|\Delta|\Sigma \vdash t[a := r] : \mathsf{T}$.

4. *Si* $\Gamma|\Delta, a :: \mathsf{R}\,|\Sigma \vdash e : \mathsf{T}$ *y* $\Gamma|\Delta|\Sigma \vdash r : \mathsf{R}$ *entonces* $\Gamma|\Delta|\Sigma \vdash e[a := r] : \mathsf{T}$.

Demostración. Inducción sobre la primera derivación de cada caso.
Algunos casos cuando $\Gamma|\cdot|\Sigma \vdash r : \mathsf{R}$ y se sustituye x por r:

- A partir de la derivación $\Gamma,\, x : \mathsf{R}\,|\,\Delta|\,\Sigma,\, \ell : \mathsf{T_2} \vdash \mathsf{mfun}_\ell\,(f.a.e) : \mathsf{T_1} \to \mathsf{T_2}$ y ocupando el lema de inversión (lema 3.2) se obtiene $\Gamma,\, x : \mathsf{R},\, f : \mathsf{T}_1 \to \mathsf{T}_2\,|\,\Delta,\, a :: \mathsf{T}_1|\,\Sigma \vdash e : \mathsf{T}_2$. Al aplicar la hipótesis de inducción para el caso de expresiones sobre esta última derivación se obtiene $\Gamma|\,\Delta|\,\Sigma,\, \ell : \mathsf{T_2} \vdash \mathsf{mfun}_\ell\,(f.a.e[x := r]) : \mathsf{T_1} \to \mathsf{T_2}$.
  Este nuevo término $\mathsf{mfun}_\ell\,(f.a.e[x := r])$ es equivalente a $(\mathsf{mfun}_\ell\,(f.a.e))[x := r]$ ya que la sustitución no afecta a los símbolos mfun_ℓ, f y a.

- De la derivación $\Gamma,\, x : \mathsf{R}\,|\,\Delta|\,\Sigma \vdash t_1\,t_2 : \mathsf{T}_2$, se obtienen por inversión las siguientes: $\Gamma,\, x : \mathsf{R}\,|\,\Delta|\,\Sigma \vdash t_1 : \mathsf{T}_1 \to \mathsf{T}_2$ y $\Gamma,\, x : \mathsf{R}\,|\,\Delta|\,\Sigma \vdash t_2 : \mathsf{T}_1$. Aplicándoles la hipótesis de inducción: $\Gamma\,|\,\Delta|\,\Sigma \vdash t_1[x := r] : \mathsf{T}_1 \to \mathsf{T}_2$ y $\Gamma\,|\,\Delta|\,\Sigma \vdash t_2[x := r] : \mathsf{T}_1$.
  Utilizando éstas se puede derivar el tipo correspondiente al término $t_1[x := r]\,t_2[x := r] = (t_1\,t_2)[x := r]$.

- La derivación $\Gamma,\, x : \mathsf{R}\,|\,\Delta|\,\Sigma \vdash \mathsf{return}\ t : \mathsf{T}$ genera, por inversión, la derivación $\Gamma,\, x : \mathsf{R}\,|\,\cdot\,|\,\Sigma \vdash t : \mathsf{T}$. Al aplicar la hipótesis de inducción a esta última derivación de tipo se obtiene $\Gamma\,|\,\cdot\,|\,\Sigma \vdash t[x := r] : \mathsf{T}$ y de esta se deriva la correspondiente a $(\mathsf{return}\,t)[x := r] = \mathsf{return}\,t[x := r]$: $\Gamma\,|\,\Delta|\,\Sigma \vdash (\mathsf{return}\,t)[x := r]\,t : \mathsf{T}$

Algunos casos cuando $\Gamma|\Delta|\Sigma \vdash r : \mathsf{R}$ y se sustituye a por r:

- De la derivación $\Gamma|\,\Delta,\, a :: \mathsf{R}\,|\,\Sigma \vdash \mathsf{case}\,(t,\, x_1.t_1,\, x_2.t_2) : \mathsf{T}$ se obtienen por inversión las siguientes: $\Gamma|\,\Delta,\, a :: \mathsf{R}\,|\,\Sigma \vdash t : \mathsf{T}_1 + \mathsf{T}_2$, $\Gamma,\, x_1 : \mathsf{T}_1|\,\Delta,\, a :: \mathsf{R}\,|\,\Sigma \vdash t_1 : \mathsf{T}$ y $\Gamma,\, x_2 : \mathsf{T}_2|\,\Delta,\, a :: \mathsf{R}\,|\,\Sigma \vdash t_2 : \mathsf{T}$. A estas tres derivaciones se les aplica la hipótesis de inducción resultando en: $\Gamma|\,\Delta\,|\,\Sigma \vdash t[a := r] : \mathsf{T}_1 + \mathsf{T}_2$, $\Gamma,\, x_1 : \mathsf{T}_1|\,\Delta\,|\,\Sigma \vdash t_1[a := r] : \mathsf{T}$ y $\Gamma,\, x_2 : \mathsf{T}_2|\,\Delta\,|\,\Sigma \vdash t_2[a := r] : \mathsf{T}$ para así derivar el siguiente juicio:

  $$\Gamma|\,\Delta\,|\,\Sigma \vdash \mathsf{case}\,(t[a := r],\, x_1.t_1[a := r],\, x_2.t_2[a := r]) : \mathsf{T}$$

  Obsérvese que el término $\mathsf{case}\,(t[a := r],\, x_1.t_1[a := r],\, x_2.t_2[a := r])$ es equivalente al término $(\mathsf{case}\,(t,\, x_1.t_1,\, x_2.t_2))[a := r]$.

- Para la derivación $\Gamma\,|\,\Delta,\, a :: \mathsf{R}\,|\,\Sigma \vdash \mathsf{let\,box}(t,\, x.e) : \mathsf{T}_2$ se obtienen por inversión las siguientes: $\Gamma|\,\Delta,\, a :: \mathsf{R}\,|\,\Sigma \vdash t : \Box\,\mathsf{T}_1$ y $\Gamma\,, x : \mathsf{T}_1|\,\Delta,\, a :: \mathsf{R}\,|\,\Sigma \vdash e : \mathsf{T}_2$.
  Aplicando la hipótesis de inducción obtenemos: $\Gamma|\,\Delta\,|\,\Sigma \vdash t[a := r] : \Box\,\mathsf{T}_1$ y
  $\Gamma\,, x : \mathsf{T}_1|\,\Delta\,|\,\Sigma \vdash e[a := r] : \mathsf{T}_2$ con las cuales se puede derivar el tipo para la expresión $\mathsf{let\,box}(t[a := r],\, x.e[a := r])$

  $$\Gamma\,|\,\Delta\,|\,\Sigma \vdash (\mathsf{let\,box}(t,\, x.e))[a := r] : \mathsf{T}_2$$

  ya que $(\mathsf{let\,box}(t,\, x.e))[a := r] = \mathsf{let\,box}(t[a := r],\, x.e[a := r])$.

□

El lema de inversión de tipos se muestra a continuación, este es muy útil en la demostración anterior y en algunas posteriores.

Lema 3.2 (Lema de inversión de tipos)**.**

- *Si* $\Gamma\,|\,\Delta|\,\Sigma \vdash x : \mathsf{T}$ *entonces* $x \in \Gamma$.
- *Si* $\Gamma\,|\,\Delta|\,\Sigma \vdash a : \mathsf{T}$ *entonces* $a \in \Delta$.
- *Si* $\Gamma\,|\,\Delta|\,\Sigma \vdash o(t_1, \ldots, t_n) : \mathsf{T}$ *entonces* $\Gamma\,|\,\Delta|\,\Sigma \vdash t_i : \mathsf{T}_i$ *con* $(1 \leq i \leq n)$ *y* $\vdash o\ : \mathsf{T}_1 \times \cdots \times \mathsf{T}_n \to \mathsf{T}$.
- *Si* $\Gamma\,|\,\Delta|\,\Sigma \vdash \star : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{Unit}$.
- *Si* $\Gamma\,|\,\Delta|\,\Sigma \vdash n : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{Int}$.
- *Si* $\Gamma\,|\,\Delta|\,\Sigma \vdash \lambda x : \mathsf{T}_1\,.t : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2$ *para algún* T_2 *con* $\Gamma,\, x : \mathsf{T}_1\,|\,\Delta|\,\Sigma \vdash t : \mathsf{T}_2$.
- *Si* $\Gamma\,|\,\Delta|\,\Sigma,\, \ell : \mathsf{T}_2 \vdash \mathsf{mfun}_\ell\,(f.a.e) : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2$ *para algunos* T_1 *y* T_2 *con* $\Gamma,\, f : \mathsf{T}_1 \to \mathsf{T}_2\,|\,\Delta,\, a :: \mathsf{T}_1|\,\Sigma \vdash e : \mathsf{T}_2$.
- *Si* $\Gamma\,|\,\Delta|\,\Sigma \vdash t_1\,t_2 : \mathsf{T}_2$ *entonces hay un tipo* T_1 *tal que* $\Gamma\,|\,\Delta|\,\Sigma \vdash t_1 : \mathsf{T}_1 \to \mathsf{T}_2$ *y* $\Gamma\,|\,\Delta|\,\Sigma \vdash t_2\ : \mathsf{T}_1$.

- *Si* $\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{inl}_{\mathsf{T}_2}\, t : \mathsf{T}_1 + \mathsf{T}_2$ *entonces* $\Gamma \mid \Delta \mid \Sigma \vdash t : \mathsf{T}_1$.
- *Si* $\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{inr}_{\mathsf{T}_1}\, t : \mathsf{T}_1 + \mathsf{T}_2$ *entonces* $\Gamma \mid \Delta \mid \Sigma \vdash t : \mathsf{T}_2$.
- *Si* $\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{case}\,(t,\, x_1.t_1,\, x_2.t_2) : \mathsf{T}$ *entonces* $\Gamma \mid \Delta \mid \Sigma \vdash t : \mathsf{T}_1 + \mathsf{T}_2$ *para algunos* T_1 *y* T_2 *y además* $\Gamma,\, x_1 : \mathsf{T}_1 \mid \Delta \mid \Sigma \vdash t_1 : \mathsf{T}$ *y* $\Gamma,\, x_2 : \mathsf{T}_2 \mid \Delta \mid \Sigma \vdash t_2 : \mathsf{T}$.
- *Si* $\Gamma \mid \Delta \mid \Sigma \vdash \langle t_1, t_2 \rangle : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{T}_1 \times \mathsf{T}_2$ *y* $\Gamma \mid \Delta \mid \Sigma \vdash t_1 : \mathsf{T}_1$, $\Gamma \mid \Delta \mid \Sigma \vdash t_2 : \mathsf{T}_2$.
- *Si* $\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{fst}\, t : \mathsf{T}_1$ *entonces* $\Gamma \mid \Delta \mid \Sigma \vdash t : \mathsf{T}_1 \times \mathsf{T}_2$ *para algún* T_2.
- *Si* $\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{snd}\, t : \mathsf{T}_2$ *entonces* $\Gamma \mid \Delta \mid \Sigma \vdash t : \mathsf{T}_1 \times \mathsf{T}_2$ *para algún* T_1.
- *Si* $\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{box}\, t : \mathsf{T}'$ *entonces* $\mathsf{T}' = \Box\, \mathsf{T}$ *y* $\Gamma \mid \cdot \mid \Sigma \vdash t : \mathsf{T}$
- *Si* $\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{return}\ t : \mathsf{T}$ *entonces* $\Gamma \mid \cdot \mid \Sigma \vdash t : \mathsf{T}$.
- *Si* $\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{let\,box}(t,\, x.e) : \mathsf{T}_2$ *entonces* $\Gamma \mid \Delta \mid \Sigma \vdash t : \Box\, \mathsf{T}_1$ *y* $\Gamma, x : \mathsf{T}_1 \mid \Delta \mid \Sigma \vdash e : \mathsf{T}_2$
- *Si* $\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{letprod}\,(t,\, a_1.a_2.e) : \mathsf{T}$ *entonces* $\Gamma \mid \Delta \mid \Sigma \vdash t : \mathsf{T}_1 \times \mathsf{T}_2$ *y* $\Gamma \mid \Delta,\, a_1 :: \mathsf{T}_1,\, a_2 :: \mathsf{T}_2 \mid \Sigma \vdash e : \mathsf{T}$
- *Si* $\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{mcase}\,(t,\, a_1.e_1,\, a_2.e_2) : \mathsf{T}$ *entonces* $\Gamma \mid \Delta \mid \Sigma \vdash t : \mathsf{T}_1 + \mathsf{T}_2$, $\Gamma \mid \Delta,\, a_1 :: \mathsf{T}_1 \mid \Sigma \vdash e_1 : \mathsf{T}$ *y* $\Gamma \mid \Delta,\, a_2 :: \mathsf{T}_2 \mid \Sigma \vdash e_2 : \mathsf{T}$

Demostración. Inmediata a partir de la definición de la relación de tipado o juicios de tipo. □

La formulación de la seguridad del sistema presentado en [2] se realiza en un lenguaje cuya semántica no incluye memoización. Se describe una traducción a un sistema sin memoización en donde es fácil demostrar la seguridad. La demostración se sigue de un teorema en donde se involucran las evaluaciones de ambos sistemas. Veamos como está formulada la seguridad del sistema que se propone en este trabajo:

Teorema 3.1.

1. *Si* $\Gamma \mid \cdot \mid \Sigma \vdash t : \mathsf{T}$ *y* $\mu, t \Downarrow^t v, \mu'$ *con* μ *inicial, entonces existe* Σ' *tal que* $\Gamma \mid \cdot \mid \Sigma' \vdash v : \mathsf{T}$.
2. *Si* $\Gamma \mid \cdot \mid \Sigma \vdash e : \mathsf{T}$ *y la evaluación* $\mu, e \Downarrow^e_{\beta@\ell} v, \mu'$ *se origina en una aplicación*[3] $\mu^\star, t_1 t_2 \Downarrow^t v^\star, \mu^{\star\prime}$ *con* $\mu^\star$ *inicial, entonces existe* Σ' *tal que* $\Gamma \mid \cdot \mid \Sigma' \vdash v : \mathsf{T}$.

Para demostrar el teorema anterior se tomará la misma idea pero con nuevas aportaciones: primero se dará una definición formal de un sistema sin efectos llamado S, después se definirá la traducción del sistema SM al sistema S que no incluye memoización en la semántica del lenguaje pero que está presente mediante mecanismos de la traslación.

[3] Es decir que $\mu, e \Downarrow^e_{\beta@\ell} v\, \mu'$ se obtiene como parte de la derivación de la premisa que involucra a $\Downarrow^e_{\bullet@\ell'}$ en la regla (Emapply).

Cabe destacar que esta traducción no aparece en [2] ni se demuestra que es fiel, en este trabajo se observarán ambas más adelante. Finalmente, como la idea principal es traducir el sistema que incluye memoización a uno que no la tenga se realiza una segunda traducción al sistema PCF que es libre de memoización.

Ahora veamos unos ejemplos del uso de la memoización para resolver problemas antes de continuar con lo descrito para demostrar la seguridad de SM.

3.2. Ejemplos

Se desarrollan diferentes ejemplos tanto en sintaxis concreta como en abstracta.

- Fibonacci
  El clásico ejemplo de Fibonacci se presenta memoizado:

```
mfib: □Int -> Int

mfun_ℓ mfib (a::□Int) = letbox x = a in
                            return (if x<2 then x
                                           else mfib (box (x-1)) +
                                                mfib (box (x-2))).
```

  El constructor let box en la definición hace que el valor contenido en a, que es un recurso, sea expuesto antes de realizar las llamadas recursivas generando un chequeo en la tabla de memoización . El dominio de la función debe ser de tipo $\square\,\mathsf{T}$ para indicar que el argumento de la función debe ser explorado, es decir, se debe hacer una exploración del tipo subyacente T dado que el resultado final depende en su totalidad de este valor.
  Veamos cómo queda bajo la sintaxis del sistema:

```
mfun_ℓ(mfib.a.letbox(a, m.e))
e = return case(mq(m,2),
                x.1,
                y.suma(mfib(box(resta(x,1))), mfib(box(resta(x,2)))))
```

- Dependencias Parciales
  Este ejemplo se adaptó de [2] donde se usa para mostrar el uso de la memoización selectiva:

```
mf: Int × (□Int × □Int) -> Int

mfun_ℓ mf (a::Int × (□Int × □Int)) = mcase (fst a)< 0 of
                                        a_1 => e_1
                                        a_2 => e_2
```

```
e1 = letbox y' = fst(snd a) in return (y'-1)
e2 = letbox z' = snd(snd a) in return (z'+1)
```

La función `mf` debe recibir tres números como argumentos, en este caso recibe una pareja que contiene al primer número que es el punto de decisión y otra pareja con los dos números restantes. El primer número no tiene tipo $\Box\,\mathsf{T}$ ya que sólo se explorará mediante mcase para iniciar la rama de exploración. En la segunda pareja cada número tiene el tipo $\Box\,\mathsf{T}$ para enfatizar que el resultado final depende de ellos. En sintaxis del sistema:

```
mfun_ℓ(mf.a.mcase(mq(fst a, 0),
                  a1.letbox(fst(snd a), y'.return resta(y', 1)),
                  a2.letbox(snd(snd a), z'.return suma(z', 1))))
```

- El problema de la mochila
  Este ejemplo se deriva del "problema de la mochila":
  Dado un conjunto de objetos, cada uno con peso y valor, se debe determinar el número de objetos que se pueden guardar dentro de una mochila tales que el peso total de los objetos no rebase el peso de la mochila pero sea el mayor posible.
  Se considera el problema **0-1** en donde cada objeto puede estar o no en la mochila, es decir no puede haber "copias" de objetos en la mochila ni partes de objetos, además de que los pesos son enteros positivos.

  Este problema se puede resolver usando programación dinámica. La solución para una serie de i objetos a escoger está entre algunas de las dos siguientes:

  - o bien puede incluir al i-ésimo en donde se debe considerar su valor v_i y la solución al problema para $i-1$ objetos junto con la disminución en el peso de la mochila en w_i
  - o bien puede no incluir al i-ésimo objeto generando una solución en donde se tienen $i-1$ objetos junto con el peso original de la mochila.

  Aquí se muestra una codificación de la solución usando recursión:

```
mks: □Int × □List(Int × Int) ->  Int

mfun_ℓ mks (a::□Int × □List(Int × Int)) =
      letbox c = (fst a) in
               letbox l = (snd a) in
                        return (if (l=nil) then 0 else t1)

t1 = if (c < fst(head y)) then (mks (box c, box (tail y))) else t2
```

```
t2 = ((λv1.λv2.if (v1 > v2) then v1 else v2)
                          (mks (box c, box (tail y))))
        (snd(head y) + mks (box (c-fst(head y)), box (tail y)))
```

Se debe suponer que el tipo lista pertenece a los tipos básicos dentro del sistema de tipos. El argumento de entrada para `mks` es un par que contiene como primer entrada el peso máximo de la mochila y como segundo argumento una lista de los objetos. Los objetos están representados por parejas formadas con el peso y el valor de cada objeto. Tanto el peso de la mochila como la lista de los objetos tienen tipo $\mathsf{box}\,\mathsf{T}$ para indicar que el resultado final depende de ambos, del peso de la mochila y de los objetos. En la sintaxis del sistema:

```
mfun_ℓ(mks.a.letbox(fst a,
                  c.letbox(snd a, l.return case(l, x.0, y.t1))))

t1 = case(mq(c, fst(head y)),
          mks (box(c), box(tail y)),
          ((λv1.λv2.case(mq(v2, v1), v1, v2))(mks (box(c), box(tail y))))
               (suma(snd(head y),
                    mks (box(resta(c, fst(head y))), box(tail(y))))))
```

- Subsecuencia común más larga
  El ejemplo que aparece en el capítulo de memoización (pág. 16) se puede reescribir de la siguiente forma:

```
lcs: □ [Int]× [Int] → Int

mfun_ℓ(lcs.a.letbox(fst a, x.mcase(x == [],
                                   w1.return 0,
                                   w2.letbox(snd a, y.e1))))

e1 = mcase(x == [],
           w3.return 0,
           w4.mcase((head x) == (head y),
                    a1.suma(1, lcs (tail x) (tail y))
                    a2.max( lcs x (tail y), lcs (tail x) y)))
```

3.3. Un sistema sin efectos

Para probar la seguridad del sistema SM se define un sistema auxiliar S que mantiene la idea de selección pero elimina los efectos es decir la memoización. Este nuevo sistema no aparece en [2], corresponde a una aportación de este trabajo. Este sistema es similar al

original pero tiene diferencias en la semántica dinámica para expresiones. A continuación se describe el nuevo sistema:

- *Tipos.* Los mismos tipos que el sistema SM :

$$\mathsf{T} \ ::= \ \mathsf{B} \mid \mathsf{T} \to \mathsf{T} \mid \mathsf{T} + \mathsf{T} \mid \mathsf{T} \times \mathsf{T} \mid \Box\, \mathsf{T}$$

- *Términos.* Los mismos términos que en SM excepto que para la declaración de las funciones memoizadas no hay una etiqueta asociada a la función:

$$\begin{array}{rcl} t, r, s & ::= & x \mid a \mid o(t, \ldots, t) \mid \star \mid n \mid \lambda x : \mathsf{T}\, .r \mid \mathsf{mfun}(f.a.e) \mid rs \mid \mathsf{inl}_{\mathsf{T}}\, r \mid \mathsf{inr}_{\mathsf{T}}\, s \mid \\ & & \mathsf{case}(r, x.s, y.t) \mid \langle r, s \rangle \mid \mathsf{fst}\, r \mid \mathsf{snd}\, r \mid \mathsf{box}\, t \end{array}$$

  Los términos $\mathsf{mfun}(f.a.e)$ son llamados funciones con nombre que reemplazan a los términos de la forma $\mathsf{mfun}_\ell(f.a.e)$ del sistema SM.

- *Expresiones.* Las mismas que en SM :

$$e \ ::= \ \mathsf{return}\ t \mid \mathsf{let\,box}\,(t,\, x.e) \mid \mathsf{letprod}\,(t,\, a_1.a_2.e) \mid \mathsf{mcase}\,(t,\, a_1.e_1,\, a_2.e_2)$$

- *Contextos.* Sólo se conservan los contextos para variables ordinarias Γ y para variables de recursos Δ.

$$\Gamma \ ::= \ \cdot \mid \Gamma, x : \mathsf{T} \qquad \Delta \ ::= \ \cdot \mid \Delta, a :: \mathsf{T}$$

- *Sistema de Tipos.* Son las mismas reglas que las usadas en SM, eliminando todas las etiquetas de contextos Σ. La regla de tipado para las funciones con nombre sustituye a la regla para las funciones memoizadas:

$$\frac{\Gamma,\, f : \mathsf{T}_1 \to \mathsf{T}_2 \,|\, \Delta,\, a :: \mathsf{T}_1 \vdash e : \mathsf{T}_2}{\Gamma \,|\, \Delta \vdash \mathsf{mfun}(f.a.e) : \mathsf{T}_1 \to \mathsf{T}_2}$$

  Esta regla de inferencia satisface las propiedades de estructura de monotonía, cambio y contracción. Más aun, dado que las inferencias son dirigidas por la sintaxis también se satisface la propiedad de inversión, esta propiedad es nombrada más adelante.

3.3.1. Semántica Dinámica

La relación de evaluación para términos es completamente análoga a la del sistema SM, sólo es necesario eliminar las memorias y modificar los valores. Para las expresiones, la relación $e \Downarrow^e_{\beta; v_f} v$ depende sólo de la rama β y del valor funcional v_f que tiene la forma $\mathsf{mfun}(f.a.e)$. Para los casos de $\mathsf{let\,box}$, $\mathsf{letprod}$ o mcase las reglas de evaluación se obtienen de aquellos casos para $\Downarrow^e_{\beta@\ell}$ reemplazando $\beta@\ell$ por $\beta; v_f$ respectivamente. Para el caso de return se necesita de una función parcial de acceso $v_f@\beta$, definida a continuación, la cual computa la expresión return que se obtiene al explorar la rama β.

Definición 3.5. *Dado un término o una expresión r y una rama β se define la función de acceso parcial $r@\beta$ como sigue:*

$$\begin{array}{rcl}
\mathsf{mfun}(f.a.e)\ @\ \beta & = & e\ @\ \beta \\
\mathsf{return}\, t\ @\ \bullet & = & \mathsf{return}\, t \\
\mathsf{let\,box}(t, x.e)\ @\ \beta\widehat{\ }!v & = & e[x := v]\ @\ \beta \\
\mathsf{letprod}(t, a_1.a_2.e)\ @\ \beta & = & e\ @\ \beta \\
\mathsf{mcase}(t, a_1.e_1, a_2.e_2)\ @\ \beta\widehat{\ }\mathsf{inl} & = & e_1\ @\ \beta \\
\mathsf{mcase}(t, a_1.e_1, a_2.e_2)\ @\ \beta\widehat{\ }\mathsf{inr} & = & e_2\ @\ \beta
\end{array}$$

La notación $\beta\widehat{\ }\varepsilon$ hace explícito el último evento ε de la rama cuyos elementos anteriores son aquellos en β.

La semántica dinámica para los términos $t \Downarrow_p^t v$ se presenta a continuación, la diferencia principal es que la regla de aplicación para las funciones con nombre cambia:

$$\frac{t_1 \Downarrow_p^t v_1, \quad \ldots \quad t_n \Downarrow_p^t v_n \quad o(v_1, \ldots, v_n) \Downarrow_p^t v}{o\ (t_1, \ldots, t_n) \Downarrow_p^t v}\ (\mathsf{Ebasicop})$$

$$\frac{}{\star \Downarrow_p^t\ \star}\ (\mathsf{Eunit}) \qquad \frac{}{n \Downarrow_p^t\ n}\ (\mathsf{Enum})$$

$$\frac{}{\lambda x : \mathsf{T}\,.t \Downarrow_p^t\ \lambda x : \mathsf{T}\,.t}\ (\mathsf{Elam})$$

$$\frac{}{\mathsf{mfun}(f.a.e) \Downarrow_p^t\ \mathsf{mfun}(f.a.e)}\ (\mathsf{Emfun})$$

$$\frac{\begin{array}{rcl} t_1 & \Downarrow_p^t & v_1 = \mathsf{mfun}(f.a.e) \\ t_2 & \Downarrow_p^t & v_2 \\ e[f, a := v_1,\, v_2] & \Downarrow_{\bullet\,;\,v_1}^e & v \end{array}}{t_1\, t_2 \Downarrow_p^t v}\ (\mathsf{Emapply})$$

$$\frac{\begin{array}{rcl} t_1 & \Downarrow_p^t & \lambda x : \mathsf{T}\,.t \\ t_2 & \Downarrow_p^t & v' \\ t[x := v'] & \Downarrow_p^t & v \end{array}}{t_1 t_2 \Downarrow_p^t\ v}\ (\mathsf{Eapply})$$

$$\frac{t \Downarrow_p^t\ v}{\mathsf{inl}\, t \Downarrow_p^t\ \mathsf{inl}\, v}\ (\mathsf{Einl}) \qquad \frac{t \Downarrow_p^t\ v}{\mathsf{inr}\, t \Downarrow_p^t\ \mathsf{inr}\, v}\ (\mathsf{Einr})$$

$$\frac{\begin{array}{rcl} t & \Downarrow_p^t & \mathsf{inl}\, v \\ t_1[x_1 := v] & \Downarrow_p^t & v_1 \end{array}}{\mathsf{case}\,(t,\, x_1.t_1,\, x_2.t_2) \Downarrow_p^t\ v_1}\ (\mathsf{Ecase-l}) \qquad \frac{\begin{array}{rcl} t & \Downarrow_p^t & \mathsf{inr}\, v \\ t_2[x_2 := v] & \Downarrow_p^t & v_2 \end{array}}{\mathsf{case}\,(t,\, x_1.t_1,\, x_2.t_2) \Downarrow_p^t\ v_2}\ (\mathsf{Ecase-r})$$

$$\frac{\begin{array}{c} t_1 \;\; \Downarrow_p^t \;\; v_1 \\ t_2 \;\; \Downarrow_p^t \;\; v_2 \end{array}}{\langle t_1, t_2\rangle \;\Downarrow_p^t\; \langle v_1, v_2\rangle} \;\; (\mathsf{Epair})$$

$$\frac{t \Downarrow_p^t \; \langle v_1,\, v_2\rangle}{\mathsf{fst}\, t \Downarrow_p^t \; v_1} \;\; (\mathsf{Efst}) \qquad \frac{t \Downarrow_p^t \; \langle v_1,\, v_2\rangle}{\mathsf{snd}\, t \Downarrow_p^t \; v_2} \;\; (\mathsf{Esnd})$$

$$\frac{t \Downarrow_p^t \; v}{\mathsf{box}\, t \Downarrow_p^t \; \mathsf{box}\, v} \;\; (\mathsf{Ebox})$$

Para la evaluación de expresiones sólo cambia la correspondiente a return, la cual hace que la evaluación del argumento t de la expresión siempre se realice. La condición auxiliar $v_f@\beta = \mathsf{return}\, t$ asegura que la rama β corresponda a la exploración del valor funcional v_f que se usa para evaluar a la expresión $\mathsf{return}\, t$:

$$\frac{\begin{array}{c} v_f@\beta = \mathsf{return} \;\; t \\ t \Downarrow_p^t v \end{array}}{\mathsf{return} \;\; t \Downarrow_{\beta;v_f}^e \;\; v} \;\; (\mathsf{Ereturn})$$

$$\frac{\begin{array}{c} t \;\; \Downarrow_p^t \;\; \mathsf{box}\, v \\ e[x := v] \;\; \Downarrow_{!v\cdot\beta;v_f}^e \;\; v' \end{array}}{\mathsf{let\,box}\,(t,\, x.e) \Downarrow_{\beta;v_f}^e \;\; v'} \;\; (\mathsf{Eletbox})$$

$$\frac{\begin{array}{c} t \;\; \Downarrow_p^t \;\; \langle v_1,\, v_2\rangle \\ e[a_1, a_2 := v_1,\, v_2] \;\; \Downarrow_{\beta;v_f}^e \;\; v \end{array}}{\mathsf{letprod}\,(t,\, a_1.a_2.e) \Downarrow_{\beta;v_f}^e \;\; v} \;\; (\mathsf{Eletprod})$$

$$\frac{\begin{array}{c} t \;\; \Downarrow_p^t \;\; \mathsf{inl}\, v \\ e_1[a_1 := v] \;\; \Downarrow_{\mathsf{inl}\cdot\beta;v_f}^e \;\; v_1 \end{array}}{\mathsf{mcase}\,(t,\, a_1.e_1,\, a_2.e_2) \Downarrow_{\beta;v_f}^e, v_1} \;\; (\mathsf{Emcase-l})$$

$$\frac{\begin{array}{c} t \;\; \Downarrow_p^t \;\; \mathsf{inr}\, v \\ e_2[a_2 := v] \;\; \Downarrow_{\mathsf{inr}\cdot\beta;v_f}^e \;\; v_2 \end{array}}{\mathsf{mcase}\,(t,\, a_1.e_1,\, a_2.e_2) \Downarrow_{\beta;v_f}^e, v_2} \;\; (\mathsf{Emcase-r})$$

3.3.2. Seguridad para S

La seguridad del sistema S resulta fácil de demostrar dado que se trata de un sistema sin efectos. Una de las aportaciones de este trabajo es mostar una prueba clara de la seguridad del sistema S que ayude a demostrar la seguridad del sistema principal SM.

Lema 3.3 (Lema de sustitución)**.** *La semántica estática del sistema* S *satisface las siguientes:*

1. *Si* $\Gamma, x : \mathsf{R} \,|\Delta \vdash t : \mathsf{T}$ *y* $\Gamma|\cdot \vdash r : \mathsf{R}$ *entonces* $\Gamma|\Delta \vdash t[x := r] : \mathsf{T}$.
2. *Si* $\Gamma, x : \mathsf{R} \,|\Delta \vdash e : \mathsf{T}$ *y* $\Gamma|\cdot \vdash r : \mathsf{R}$ *entonces* $\Gamma|\Delta \vdash e[x := r] : \mathsf{T}$.
3. *Si* $\Gamma|\Delta, a :: \mathsf{R} \vdash t : \mathsf{T}$ *y* $\Gamma|\Delta \vdash r : \mathsf{R}$ *entonces* $\Gamma|\Delta \vdash t[a := r] : \mathsf{T}$.
4. *Si* $\Gamma|\Delta, a :: \mathsf{R} \vdash e : \mathsf{T}$ *y* $\Gamma|\Delta \vdash r : \mathsf{R}$ *entonces* $\Gamma|\Delta \vdash e[a := r] : \mathsf{T}$.

Demostración. Inducción sobre la primera derivación de cada caso. □

Lema 3.4 (Lema de inversión de tipos)**.**

- *Si* $\Gamma \,|\, \Delta \vdash x : \mathsf{T}$ *entonces* $x \in \Gamma$.
- *Si* $\Gamma \,|\, \Delta \vdash a : \mathsf{T}$ *entonces* $a \in \Delta$.
- *Si* $\Gamma \,|\, \Delta \vdash o(t_1, \ldots, t_n) : \mathsf{T}$ *entonces* $\Gamma \,|\, \Delta \vdash t_i : \mathsf{T}_i$ *con* $(1 \leq i \leq n)$ *y* $\vdash o \; : \mathsf{T}_1 \times \cdots \times \mathsf{T}_n \to \mathsf{T}$.
- *Si* $\Gamma \,|\, \Delta \vdash \star : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{Unit}$.
- *Si* $\Gamma \,|\, \Delta \vdash n : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{Int}$.
- *Si* $\Gamma \,|\, \Delta \vdash \lambda x : \mathsf{T}_1 . t : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2$ *para algún* T_2 *con* $\Gamma, x : \mathsf{T}_1 \,|\, \Delta \vdash t : \mathsf{T}_2$.
- *Si* $\Gamma \,|\, \Delta \vdash \mathsf{mfun}\,(f.a.e) : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{T}_1 \to \mathsf{T}_2$ *para algunos* T_1 *y* T_2 *con* $\Gamma, f : \mathsf{T}_1 \to \mathsf{T}_2 \,|\, \Delta, a :: \mathsf{T}_1 \vdash e : \mathsf{T}_2$.
- *Si* $\Gamma \,|\, \Delta \vdash t_1\, t_2 : \mathsf{T}_2$ *entonces hay un tipo* T_1 *tal que* $\Gamma \,|\, \Delta \vdash t_1 : \mathsf{T}_1 \to \mathsf{T}_2$ *y* $\Gamma \,|\, \Delta \vdash t_2 \; : \mathsf{T}_1$.
- *Si* $\Gamma \,|\, \Delta \vdash \mathsf{inl}_{\mathsf{T}_2}\, t : \mathsf{T}_1 + \mathsf{T}_2$ *entonces* $\Gamma \,|\, \Delta \vdash t : \mathsf{T}_1$.
- *Si* $\Gamma \,|\, \Delta \vdash \mathsf{inr}_{\mathsf{T}_1}\, t : \mathsf{T}_1 + \mathsf{T}_2$ *entonces* $\Gamma \,|\, \Delta \vdash t : \mathsf{T}_2$.
- *Si* $\Gamma| \,\Delta \vdash \mathsf{case}\,(t,\, x_1.t_1,\, x_2.t_2) : \mathsf{T}$ *entonces* $\Gamma| \,\Delta \vdash t : \mathsf{T}_1 + \mathsf{T}_2$ *para algunos* T_1 *y* T_2 *y además* $\Gamma,\, x_1 : \mathsf{T}_1| \,\Delta \vdash t_1 : \mathsf{T}$ *y* $\Gamma,\, x_2 : \mathsf{T}_2| \,\Delta \vdash t_2 : \mathsf{T}$.
- *Si* $\Gamma \,|\, \Delta \vdash \langle t_1, t_2 \rangle : \mathsf{T}$ *entonces* $\mathsf{T} = \mathsf{T}_1 \times \mathsf{T}_2$ *y* $\Gamma \,|\, \Delta \vdash t_1 \; : \mathsf{T}_1$, $\Gamma \,|\, \Delta \vdash t_2 : \mathsf{T}_2$.
- *Si* $\Gamma \,|\, \Delta \vdash \mathsf{fst}\, t : \mathsf{T}_1$ *entonces* $\Gamma \,|\, \Delta \vdash t \; : \mathsf{T}_1 \times \mathsf{T}_2$ *para algún* T_2.
- *Si* $\Gamma \,|\, \Delta \vdash \mathsf{snd}\, t : \mathsf{T}_2$ *entonces* $\Gamma \,|\, \Delta \vdash t \; : \mathsf{T}_1 \times \mathsf{T}_2$ *para algún* T_1.
- *Si* $\Gamma \,|\, \Delta \vdash \mathsf{box}\, t : \mathsf{T}'$ *entonces* $\mathsf{T}' = \Box\, \mathsf{T}$ *y* $\Gamma \,|\, \cdot \vdash t : \mathsf{T}$
- *Si* $\Gamma| \,\Delta \vdash \mathsf{return}\; t : \mathsf{T}$ *entonces* $\Gamma| \,\cdot \vdash t : \mathsf{T}$.

- *Si* $\Gamma|\,\Delta \vdash \mathsf{let\,box}(t,\, x.e) : \mathsf{T}_2$ *entonces* $\Gamma|\,\Delta \vdash t : \Box\,\mathsf{T}_1$ *y* $\Gamma\,, x : \mathsf{T}_1|\,\Delta \vdash e : \mathsf{T}_2$

- *Si* $\Gamma|\,\Delta \vdash \mathsf{letprod}\,(t,\, a_1.a_2.e) : \mathsf{T}$ *entonces* $\Gamma|\,\Delta \vdash t : \mathsf{T}_1 \times \mathsf{T}_2$ *y* $\Gamma|\,\Delta,\, a_1 :: \mathsf{T}_1,\, a_2 :: \mathsf{T}_2 \vdash e : \mathsf{T}$

- *Si* $\Gamma|\,\Delta|\,\Sigma \vdash \mathsf{mcase}\,(t,\, a_1.e_1,\, a_2.e_2) : \mathsf{T}$ *entonces* $\Gamma|\,\Delta|\,\Sigma \vdash t : \mathsf{T}_1 + \mathsf{T}_2$, $\Gamma|\,\Delta,\, a_1 :: \mathsf{T}_1|\,\Sigma \vdash e_1 : \mathsf{T}$ *y* $\Gamma|\,\Delta,\, a_2 :: \mathsf{T}_2|\,\Sigma \vdash e_2 : \mathsf{T}$

Demostración. Inmediata a partir de la definición de la relación de tipado o juicios de tipo. □

Con ayuda de los lemas anteriores se puede probar el teorema de seguridad del sistema S:

Teorema 3.2. *El sistema* S *es seguro, esto es:*

1. *Si* $\Gamma|\cdot \vdash t : \mathsf{T}$ *y* $t \Downarrow_p^t v$ *entonces* $\Gamma|\cdot \vdash v : \mathsf{T}$.

2. *Si* $\Gamma|\cdot \vdash e : \mathsf{T}$ *y* $e \Downarrow_{\beta;v_f}^e v$, *para cualquier* β *y* v_f, *entonces* $\Gamma|\cdot \vdash v : \mathsf{T}$.

Demostración. Se probarán ambas partes simultáneamente al realizar inducción en las relaciones de evaluación $\Downarrow_p^t$ y $\Downarrow_{\beta;v_f}^e$.
Se muestran los casos para la aplicación de funciones con nombre y para la expresión return.

- Considerar que $t = t_1 t_2$ con $t_1 t_2 \Downarrow_p^t v$ derivado de $t_1 \Downarrow_p^t v_1 = \mathsf{mfun}(f.a.e)$, $t_2 \Downarrow_p^t v_2$ y $e[f, a := v_1, v_2] \Downarrow_{\bullet\,;v_1}^e v$. Dado que $\Gamma|\cdot \vdash t_1 t_2 : \mathsf{T}$, por el lema de inversión se tiene que $\Gamma|\cdot \vdash t_1 : \mathsf{R} \to \mathsf{T}$ y $\Gamma|\cdot \vdash t_2 : \mathsf{R}$. Por hipótesis de inducción de la parte (i) se tiene que $\Gamma|\cdot \vdash v_1 : \mathsf{R} \to \mathsf{T}$ y $\Gamma|\cdot \vdash v_2 : \mathsf{T}$. Observemos que el tipo de v_1 implica que $\Gamma, f : \mathsf{R} \to \mathsf{T}\,|a :: \mathsf{R} \vdash e : \mathsf{T}$. Por lo tanto por el lema de sustitución partes (ii) y (iv) obtenemos que $\Gamma|\cdot \vdash e[f, a := v_1, v_2] : \mathsf{T}$. Finalmente como $e[f, a := v_1, v_2] \Downarrow_{\bullet\,;v_1}^e v$, la hipótesis de inducción parte (ii) de este teorema implica que $\Gamma|\cdot \vdash v : \mathsf{T}$.

- Considerar que $e = \mathsf{return}\,t$ con $\mathsf{return}\,t \Downarrow_{\beta;v_f}^e v$ derivado de $v_f @ \beta = \mathsf{return}\,t$ y $t \Downarrow_p^t v$.
  Se tiene que $\Gamma|\cdot \vdash \mathsf{return}\,t : \mathsf{T}$, y por el lema de inversión de tipos, $\Gamma|\cdot \vdash t : \mathsf{T}$ la cual por hipótesis de inducción parte (i), junto con $\mathsf{return}\,t \Downarrow_{\beta;v_f}^e v$ se obtiene $\Gamma|\cdot \vdash v : \mathsf{T}$.

□

Este sistema va a ser traducido al sistema PCF ya que en [2] no hay un sistema realmente puro en donde se pueda demostrar que la memoización no afecta el resultado de la función que hace uso de ella. Primero veamos la traducción a un sistema que no incluye memoización semánticamente para poder demostrar la seguridad de SM.

3.4. Traducción de SM a S

Esta sección está dedicada a desarrollar una traducción fiel desde SM a S para probar la seguridad del primero. Para ello se necesitan las siguientes definiciones:

Definición 3.6. *La traducción $(\cdot)^-$ de términos y tipos de* SM *a términos y tipos de* S *se define como sigue:*

- *La traducción en tipos es la función identidad.*
- *La traducción en términos y expresiones es el mapeo que borra etiquetas:*

r	r^-
x	x
a	a
$o(t, \ldots, t)$	$o(t^-, \ldots, t^-)$
$\star$	$\star$
n	n
$\lambda x : \mathsf{T}\,.r$	$\lambda x : \mathsf{T}\,.r^-$
$\mathsf{mfun}_\ell(f.a.e)$	$\mathsf{mfun}(f.a.e^-)$
rs	r^-s^-
$\mathsf{inl}_\mathsf{T}\, r$	$\mathsf{inl}_\mathsf{T}\, r^-$
$\mathsf{inr}_\mathsf{T}\, s$	$\mathsf{inr}_\mathsf{T}\, s^-$
$\mathsf{case}(r, x.s, y.t)$	$\mathsf{case}(r^-, x.s^-, y.t^-)$
$\langle r, s\rangle$	$\langle r^-, s^-\rangle$
$\mathsf{fst}\, r$	$\mathsf{fst}\, r^-$
$\mathsf{snd}\, r$	$\mathsf{snd}\, r^-$
$\mathsf{box}\, t$	$\mathsf{box}\, t^-$
$\mathsf{return}\, t$	$\mathsf{return}\, t^-$
$\mathsf{let\,box}(t,\ x.e)$	$\mathsf{let\,box}(t^-,\ x.e^-)$
$\mathsf{letprod}(t,\ a_1.a_2.e)$	$\mathsf{letprod}(t^-,\ a_1.a_2.e^-)$
$\mathsf{mcase}(t,\ a_1.e_1,\ a_2.e_2)$	$\mathsf{mcase}(t^-,\ a_1.e_1^-,\ a_2.e_2^-)$

Más adelante se aplicará esta traducción a las ramas β, obteniendo las ramas β^- al reemplazar cada evento de la forma $!v$ que aparece en β, por $!(v^-)$.

Características importantes de la traslación son la compatibilidad con sustitución y con las derivaciones de tipos.

Lema 3.5. *Si r es un término o una expresión entonces $r[x := t]^- = r^-[x := t^-]$ y $r[a := t]^- = r^-[a := t^-]$.*

Demostración. Inducción simultánea en términos y expresiones. □

Proposición 3.1. *La traducción $t \mapsto t^-$ respeta los tipos.*

1. *Si* $\Gamma|\Delta|\Sigma \vdash t : \mathsf{T}$ *entonces* $\Gamma|\Delta \vdash t^- : \mathsf{T}$.

2. *Si* $\Gamma|\Delta|\Sigma \vdash e : \mathsf{T}$ *entonces* $\Gamma|\Delta \vdash e^- : \mathsf{T}$.

Demostración. Inducción simultánea en las derivaciones de tipos. Se mostrarán los casos importantes:

- Sea $t = \mathsf{mfun}_\ell(f.a.e)$, su derivación de tipo es: $\Gamma \mid \Delta \mid \Sigma \vdash \mathsf{mfun}_\ell\,(f.a.e) : \mathsf{T}_1 \to \mathsf{T}_2$
  Por el lema de inversión se obtiene: $\Gamma,\ f : \mathsf{T}_1 \to \mathsf{T}_2 \mid \Delta,\ a :: \mathsf{T}_1 \mid \Sigma \vdash e : \mathsf{T}_2$.
  Aplicando la hipótesis de inducción para el caso de expresiones se obtiene: $\Gamma,\ f : \mathsf{T}_1 \to \mathsf{T}_2 \mid \Delta,\ a :: \mathsf{T}_1 \vdash e^- : \mathsf{T}_2$. A partir de este juicio se puede derivar el tipo de $t^- = \mathsf{mfun}(f.a.e^-)$ que es $\Gamma \mid \Delta \vdash \mathsf{mfun}\,(f.a.e^-) : \mathsf{T}_1 \to \mathsf{T}_2$.

- Para el caso en que $e = \mathsf{return}\,t$ la derivación de tipo correspondiente es $\Gamma|\,\Delta \mid \Sigma \vdash \mathsf{return}\ t : \mathsf{T}$ y por inversión se tiene $\Gamma|\ \cdot\ |\,\Sigma \vdash t : \mathsf{T}$.
  Aplicando la hipótesis de inducción a esta última obtenemos la derivación $\Gamma|\,\cdot \vdash t^- : \mathsf{T}$ con la cual se puede derivar $\Gamma|\,\Delta \vdash (\mathsf{return}\,t)^- : \mathsf{T}$ que es el juicio de $e^- = \mathsf{return}\,t^-$.

□

Para probar que la traducción es fiel se necesita simular una memoria μ de un modo adecuado. Para este propósito se usan tablas de funciones que asocian etiquetas de locaciones con valores de funciones con nombre.

Definición 3.7. *Una tabla de funciones* τ *es una función parcial* $\tau : \mathcal{L} \to \mathcal{F}$ *con dominio finito, mapeando etiquetas de locaciones a valores funcionales de la forma* $\mathsf{mfun}(f.a.e)$.

La justificación de la regla (Enotfound) se realiza mediante el siguiente

Lema 3.6. *Si* $\mu, t \Downarrow^t v, \mu'$, $\mu(\ell)@\beta = \mathsf{return}\,t$ *y* $\mu(\ell)(\beta)$ *está indefinida entonces* $\mu'(\ell)(\beta)$ *también está indefinida.*

Demostración. Véase la prueba que aparece en [1] en donde se usan las tablas de función y funciones de acceso. □

Para que la simulación de la evaluación sea completa, se necesita asociar la tabla de funciones τ con la memoria μ. La relación entre la memoria μ en el sistema SM y la tabla τ en el sistema S es la siguiente: para cada función memoizada f que tiene relacionada una tabla de memoización almacenada en μ bajo una única etiqueta ℓ, existe una función con nombre f^- que es la imagen de la misma locación ℓ en τ. Esta idea se refleja en la siguiente

Definición 3.8. *Sea* μ *una memoria,* τ *una tabla de funciones,* r *un término o expresión y* e *una expresión.*

- τ *es consistente con* r *si y sólo si para cada subtérmino de* r *de la forma* $\mathsf{mfun}_\ell(f.a.e)$, *se tiene* $\tau(\ell) = \mathsf{mfun}(f.a.e^-)$

- τ *es consistente con* μ *si y sólo si para toda* $\ell \in dom(\mu)$ *y* $\beta \in dom(\mu(\ell))$, *si* $\mu(\ell)(\beta) = v$ *entonces* τ *es consistente con* v.

- τ *es compatible con* μ *si y sólo si:*

  - $dom(\mu) = dom(\tau)$
  - τ *es consistente con* μ.
  - *Para toda* $\ell \in dom(\mu)$, $\beta \in dom(\mu(\ell))$ *y* t *término, si* $\mu(\ell)(\beta) = v$ *y* $\tau(\ell)@\beta^- = \mathsf{return}\, t^-$ *entonces* $t^- \Downarrow_p^t v^-$. *Llamemos a esta condición* $(\diamond)$.

Para probar que la traducción simula la relación de evaluación se necesita de los siguientes conceptos:

Definición 3.9. *Una rama aumentada* γ *es una lista de eventos aumentados* ϵ. *Un evento aumentado guarda los puntos de decisión y los relaciona con las variables de recurso.*

$$\begin{array}{rcl} \gamma & ::= & \bullet \mid \epsilon \cdot \gamma \\ \epsilon & ::= & (v) \mid\, !v \mid \langle v_1, v_2 \rangle \mid\, \mathsf{inl}\, v \mid\, \mathsf{inr}\, v \end{array}$$

Las funciones parciales de acceso para las ramas aumentadas se definen como sigue:

$$\begin{array}{rcl} \mathsf{mfun}_\ell(f.a.e) \mathbin{@} \gamma\hat{}(v) & = & e[f, a := \mathsf{mfun}_\ell(f.a.e), v] \mathbin{@} \gamma \\ e \mathbin{@} \bullet & = & e \\ \mathsf{let\,box}(t, x.e) \mathbin{@} \gamma\hat{}!v & = & e[x := v] \mathbin{@} \gamma \\ \mathsf{letprod}(t, a_1.a_2.e) \mathbin{@} \gamma\hat{}\langle v_1, v_2 \rangle & = & e[a_1, a_2 := v_1, v_2] \mathbin{@} \gamma \\ \mathsf{mcase}(t, a_1.e_1, a_2.e_2) \mathbin{@} \gamma\hat{}\,\mathsf{inl}\, v & = & e_1[a_1 := v] \mathbin{@} \gamma \\ \mathsf{mcase}(t, a_1.e_1, a_2.e_2) \mathbin{@} \gamma\hat{}\,\mathsf{inr}\, v & = & e_2[a_1 := v] \mathbin{@} \gamma \end{array}$$

Las ramas aumentadas son usadas dentro de la demostración de la proposición 3.2 que aparece más adelante.

Definición 3.10. *Dada una rama aumentada* γ *se obtiene una rama simple denotada por* γ° *al olvidar los eventos de la forma* (v) *y* $\langle v_1, v_2 \rangle$ *así como los eventos* v *en las inyecciones* $\mathsf{inl}\, v$ *e* $\mathsf{inr}\, v$.
Además, la traducción γ^-, *de una rama aumentada* γ, *se obtiene al reemplazar cada valor* v *que aparece en* γ *por* v^-.

Operacionalmente, las ramas aumentadas generan el mismo resultado que las ramas simples como lo asegura el siguiente

Lema 3.7. *Las ramas aumentadas no modifican la expresión return. Esto es, si* $\tau(\ell)@\gamma = \mathsf{return}\, t$ *entonces* $\tau(\ell)@\gamma^\circ = \mathsf{return}\, t$.

Demostración. Véase la demostración que aparece en [1]. □

La siguiente proposición muestra que la preservación de la semántica en la memoización selectiva del sistema SM es la misma respecto a la semántica selectiva sin memoización del sistema S y permite concluir que la traducción es fiel. Esta proposición está reformulada de la que aparece en [2] para hacerla clara respecto a la traducción.

Proposición 3.2. *La traducción $t \mapsto t^-$ simula la evaluación bajo las siguientes condiciones:*

1. *Sea τ consistente con t y compatible con la memoria μ. Si $\mu, t \Downarrow^t v, \mu'$ entonces $t^- \Downarrow_p^t v^-$ y existe $\tau' \supseteq \tau$ tal que τ' es consistente con v y compatible con μ'.*

2. *Sea τ consistente con la expresión e y compatible con la memoria μ, β sea una rama simple y γ una rama aumentada. Si $\mu, e \Downarrow^e_{\beta@\ell} v, \mu'$, $\gamma^\circ = \beta$ y $\tau(\ell)@\gamma^- = e^-$ entonces existe $\tau' \supseteq \tau$ tal que τ' es consistente con v y compatible con μ' y $e^- \Downarrow^e_{\beta\,;\,\tau'(\ell)} v^-$*

Demostración. Las dos partes son demostradas simultáneamente por inducción sobre las dos relaciones de evaluación. Se muestran los casos más relevantes:

- $t = \langle t_1, t_2\rangle$ y μ, $\langle t_1, t_2\rangle \Downarrow^t \langle v_1, v_2\rangle$, μ' derivada de la regla Epair.
  La tabla de funciones τ es consistente con t_1 y con t_2 ya que lo es para el par. Entonces, por hipótesis de inducción existe τ_1 consistente con v_1 y adecuada para μ_1, también es consistente con t_2 ya que τ lo era.
  Así mismo por hipótesis de inducción existe $\tau_2 \supseteq \tau_1$ consistente con v_2 y adecuada para μ_2. Finalmente τ_2 es consistente con v_1 y v_2 y así es consistente con $\langle v_1, v_2\rangle$.

- $t = t_1 t_2$ y μ, $t_1 t_2 \Downarrow^t v$, μ' derivada de $\mu, t_1 \Downarrow^t v_1 = \mathsf{mfun}_\ell(f.a.e), \mu_1$, $\mu_1, t_2 \Downarrow^t v', \mu_2$ y $\mu_2, e[f, a := v_1, v_2] \Downarrow^e_{\bullet@\ell} v, \mu'$.
  Sea τ consistente con t y compatible con μ. En particular τ es consistente con t_1, de la cual por hipótesis de inducción se obtiene que $t_1^- \Downarrow_p^t v_1^-$ y $\tau_1 \supseteq \tau$ consistente con v_1 y compatible con μ_1. Aplicando la hipótesis de inducción a t_2 con τ_1, que es posible ya que τ_1 extiende a τ y por tanto τ_1 también es consistente con t_2, obtenemos que $t_2^- \Downarrow_p^t v_2^-$ y $\tau_2 \supseteq \tau_1$ consistente con v_2 y compatible con μ_2. Obsérvese que la tabla de funciones τ_2 también es consistente con $e[f, a := v_1, v_2]$ ya que τ_2 es consistente con v_1, v_2 y también con e para e subexpresión de v_1.
  Falta mostrar una γ tal que $\gamma^\circ = \bullet$ y $\tau_2(\ell)@\gamma^- = (e[f, a := v_1, v_2])^-$. De la hipótesis de inducción para $e[f, a := v_1, v_2]$ se tiene a τ tal que $e[f, a := v_1, v_2]^- \Downarrow^e_{\bullet\,;\,\tau'(\ell)} v^-$ que genera a $t^- \Downarrow_p^t v^-$ como se desea. Definir $\gamma = (v_2) \cdot \bullet$, entones se tiene:

$$\begin{aligned} \tau_2(\ell)@\gamma^- &= v_1^-@\gamma^- \\ &= e^-[f, a := v_1^-, v_2^-] \\ &= (e[f, a := v_1, v_2])^- \ \text{(por lema 3.5)} \end{aligned}$$

- $e = \mathsf{return}\, t$ y $\mu, e \Downarrow^e_{\beta@\ell} v, \mu'$ derivada de la regla (found).
  De esta regla se tiene que $\mu = \mu'$ y $\mu(\ell)(\beta) = v$. Suponer que τ es consistente con $\mathsf{return}\, t$ y compatible con μ, $\gamma^\circ = \beta$ y $\tau(\ell)@\gamma^- = (\mathsf{return}\, t)^- = \mathsf{return}\, t^-$. Observemos

que como $\mu(\ell)(\beta) = v$ la compatibilidad de τ con μ implica que τ es consistente con v. Más aún como $\tau(\ell)@\gamma^- = \mathsf{return}\, t^-$, el lema 3.7 implica que $\tau(\ell)@(\gamma^-)^\circ = \mathsf{return}\, t^-$. Pero $(\gamma^-)^\circ = (\gamma^\circ)^- = \beta^-$, por lo tanto $\tau(\ell)@\beta^- = \mathsf{return}\, t^-$ y nuevamente por la compatibilidad se tiene que $t^- \Downarrow_p^t v^-$. Finalmente al utilizar la regla de (Ereturn) se tiene $\mathsf{return}\, t^- \Downarrow^e_{\beta\,;\,\tau(\ell)} v^-$. Así en este caso es suficiente que se tome $\tau' = \tau$.

- $e = \mathsf{return}\, t$ y $\mu, e \Downarrow^e_{\beta@\ell} v, \mu'$ derivada de la regla (notfound).
  Suponer que τ es consistente con $\mathsf{return}\, t$ y compatible con μ, $\gamma^\circ = \beta$ y $\tau(\ell)@\gamma^- = (\mathsf{return}\, t)^- = \mathsf{return}\, t^-$. Por hipótesis de inducción existe $\tau_1 \supseteq \tau$ consistente con v y compatible con μ' y $t^- \Downarrow_p^t v^-$. Tomar $\tau' = \tau_1$, sólo resta demostrar que τ' es compatible con $\mu'' = \mu'[\ell \leftarrow \theta'[\beta \mapsto v]]$:

  - $dom(\mu'') = dom(\tau')$ ya que μ'' es sólo una actualización de μ'.
  - τ' es consistente con μ''. Obsérvese que, como τ' es consistente con μ' es suficiente con mostrar la consistencia de la actualización $\mu''(\ell)(\beta) = v$. En este caso se necesita mostrar que τ' es consistente con v, pero ya lo es por la definición de τ'.
  - Para mostrar la condición ($\diamond$) se cumple, es suficiente demostrar que para ℓ, β y un término arbitrario t' tal que $\mu''(\ell)(\beta) = v$ y $\tau(\ell)@\beta^- = \mathsf{return}\, t'^-$. Pero se tiene que $\tau(\ell)@\beta^- = \tau(\ell)@\gamma^- = \mathsf{return}\, t^-$ lo cual implica que $t = t'$, para $\_@\_$ función. Falta mostrar que $t^- \Downarrow_p^t v^-$ pero esto es consecuencia de la hipótesis de inducción.

□

Es fácil ver que si $t = f\, v$ es una aplicación donde la función f no tiene variables libres, la parte (i) de la proposición 3.2 implica que la traducción de una función memoizada calcula la aplicación $t^- = f^-\, v^-$ que corresponde a una función no memoizada con el mismo resultado que f bajo el argumento v.

A continuación se dan las condiciones que garantizan la existencia de una tabla de función como lo requiere la proposición 3.2.

Lema 3.8. *Si $(e \mathbin{@} \gamma)$ está definida, entonces para cada evento aumentado ϵ*

$$(e \mathbin{@} \gamma) \mathbin{@} \epsilon \;=\; e \mathbin{@} (\epsilon \cdot \gamma)$$

Demostración. Inducción sobre γ

1. Caso Base $\gamma = \bullet$
   De la definición $e \mathbin{@} \bullet = e$ por lo tanto $(e \mathbin{@} \bullet)\downarrow$, entonces $e \mathbin{@} \bullet \mathbin{@} \epsilon = e \mathbin{@} \epsilon = e \mathbin{@} \epsilon \cdot \bullet$.

2. Caso Inductivo $\gamma = \gamma_1 \hat{}\ \epsilon_1$
   Suponer que $(e \mathbin{@} \gamma)\downarrow$ entonces $(e \mathbin{@} \gamma_1 \hat{}\ \epsilon_1)\downarrow$ así se obtiene que e y ϵ_1 son compatibles de acuerdo a la definición recursiva para la función de acceso parcial y en tal caso $e \mathbin{@} \gamma_1 \hat{}\ \epsilon_1 = e' \mathbin{@} \gamma_1$ donde e' se obtuvo al aplicar la definición de la función @ a e y ϵ_1. Por lo tanto $(e' \mathbin{@} \gamma_1)\downarrow$ pues $(e \mathbin{@} \gamma_1 \hat{}\ \epsilon_1)\downarrow$.

De la hipótesis de inducción tenemos que: $(e' \mathbin{@} \gamma_1) \mathbin{@} \epsilon = e' \mathbin{@} \epsilon \cdot \gamma_1$.
Ahora tenemos que:

$$\begin{aligned} e \mathbin{@} \epsilon \cdot \gamma &= e \mathbin{@} \epsilon \cdot (\gamma_1 \hat{} \epsilon_1) \\ &= e \mathbin{@} (\epsilon \cdot \gamma_1) \hat{} \epsilon_1 \\ &= e' \mathbin{@} \epsilon \cdot \gamma_1 \\ &= (e' \mathbin{@} \gamma_1) \mathbin{@} \epsilon \end{aligned}$$

Por otra parte:

$$\begin{aligned} (e \mathbin{@} \gamma) \mathbin{@} \epsilon &= (e \mathbin{@} \gamma_1 \hat{} \epsilon_1) \mathbin{@} \epsilon \\ &= (e' \mathbin{@} \gamma_1) @ \epsilon \end{aligned}$$

□

Lema 3.9. *Sea e una expresión tal que* $\mu,\ e \Downarrow^e_{\beta@\ell} v,\ \mu'$ *se origina en una aplicación* $\mu^\star, t_1 t_2 \Downarrow^t_p v^\star, \mu^{\star\prime}$ *con* $\mu^\star$ *inicial, entonces existe una tabla de funciones* τ_e *y una rama aumentada* γ *tal que* τ_e *es consistente con e y compatible con* μ, $\gamma^\circ = \beta$ *y* $\tau_e(\ell)@\gamma^- = e^-$.

Demostración. De acuerdo al lema se tiene la siguiente situación:

$$\dfrac{\mu^\star,\ t_1 \Downarrow^t v_1 = \mathsf{mfun}_\ell(f.a.e_1), \mu_1 \qquad \mu_1,\ t_2 \Downarrow^t v_2, \mu_2 \qquad \dfrac{\mu, e \Downarrow^e_{\beta@\ell} v, \mu'}{\mu'',\ e' \Downarrow^e_{\beta_1@\ell} v,\ \mu'} \quad \vdots \quad \mu_2, e_1[f, a := v_1, v_2] \Downarrow^e_{\bullet@\ell} v^\star,\ \mu^{\star\prime}}{\mu^\star,\ t_1 t_2 \Downarrow^t v^\star,\ \mu^{\star\prime}}$$

La prueba procederá por inducción en el número n de las reglas de evaluación desde $e_1[f, a := v_1, v_2]$ hasta la evaluación de e.

- Base de la inducción: $n = 0$ que significa que no hay regla de inferencia, es decir, $e = e_1[f, a := v_1, v_2]$ y $v_1 = \mathsf{mfun}_\ell(f.a.e_1)$. Como $\mu^\star$ es inicial se puede construir τ consistente con $t_1 t_2$ y compatible con $\mu^\star$. De esta tabla, τ, aplicaciones repetidas de la proposición 3.2 obtienen la tabla buscada τ_e. En particular se tiene que $\tau_e(\ell) = \mathsf{mfun}(f.a.e_1^-)$.
  Definir ahora $\gamma = (v_2) \cdot \bullet$, y observar que $\tau_e(\ell)@\gamma^- = e_1^-[f, a := v_1^-, v_2^-] \cdot \bullet = e_1^-[f, a := v_1^-, v_2^-]$. Pero por el lema 3.5 se tiene que $e_1^-[f, a := v_1^-, v_2^-] = e_1[f, a := v_1, v_2]^- = e^-$ para completar esta parte.

- Paso inductivo: asumir que hay $n + 1$ reglas de inferencia desde $e_1[f, a := v_1, v_2]$ hasta e. La prueba se hará por los casos posibles de la regla superior.

  - Para el caso de $\mathsf{let\,box}$ se tiene $e' = \mathsf{let\,box}(t, x.e_1),\ e = e_1[x := v_1], \beta = !v_1 \cdot \beta_1$ y $\mu'', t \Downarrow^t_p \mathsf{box}\, v_1, \mu$. Por hipótesis de inducción existen $\tau_{e'}$ y γ_1 tales que $\tau_{e'}$ es consistente con e' y compatible con $\mu'', \tau_{e'}(\ell)@\gamma_1^- = e'^-$ y $\gamma_1^\circ = \beta_1$. En particular $\tau_{e'}$ es consistente con t y por lo tanto, por la proposición 3.2, existe τ'' tal que $\tau'' \supseteq \tau_{e'}$ es consistente con $\mathsf{box}\, v_1$ y compatible con μ. Más aún, τ'' también es

consistente con $e = e_1[x := v_1]$ y como es compatible con μ se puede definir $\tau_e = \tau''$.
Definir $\gamma = !v_1 \cdot \gamma_1$ y observar que $\gamma^\circ = !v_1 \cdot \gamma_1^\circ = !v_1 \cdot \beta_1 = \beta$, suficiente para mostrar que $\tau_e(\ell)@\gamma^- = e^-$.

$$\begin{array}{rcl} \tau_e(\ell)@\gamma^- & = & \tau_{e'}(\ell)@\gamma^- \\ & = & \tau_{e'}(\ell)@(!v_1^- \cdot \gamma_1^-) \\ & = & (\tau_{e'}(\ell)@\gamma_1^-)@!v_1^- \quad (\text{ por lema 3.8}) \\ & = & \mathsf{let\,box}(t^-, x.e_1^-)@!v_1^- \\ & = & e_1^-[x := v_1^-]@\bullet \\ & = & e_1^-[x := v_1^-] = e^- \quad (\text{por lema 3.5}) \end{array}$$

- $e' = \mathsf{letprod}(t, a_1.a_2.e_1)$ entonces $e = e_1[a_1, a_2 := v_1, v_2]$ de donde sabemos que t se redujo a $\langle v_1, v_2\rangle$ y $\beta = \beta_1$.
  Por hipótesis de inducción existen $\tau_{e'}$ y γ_1 tales que $\tau_{e'}$ es consistente con e' y es adecuada para μ_1, $\gamma_1^\circ = \beta_1$ y $\tau_{e'}(\ell)@\gamma_1^- = e'^-$.
  Sean $\tau_e(\ell) = \tau_{e'}(\ell)$ y $\gamma = (v_1, v_2) \circ \gamma_1$, así $\gamma^\circ = \gamma_1^\circ = \beta = \beta_1$ entonces:

$$\begin{array}{rcl} \tau_e(\ell)@\gamma^- & = & (\tau_{e'}(\ell)@\gamma)^- \\ & = & (\tau_{e'}(\ell)@\langle v_1, v_2\rangle \cdot \gamma_1)^- \\ & = & ((\tau_{e'}(\ell)@\gamma_1)@\langle v_1, v_2\rangle)^- \\ & = & (\mathsf{letprod}(t, a_1.a_2.e_1)@\langle v_1, v_2\rangle \circ \bullet)^- \\ & = & (e_1[a_1, a_2 := v_1, v_2]@\bullet)^- \\ & = & e_1^-[a_1, a_2 := v_1^-, v_2^-] = e^- \end{array}$$

- $e' = \mathsf{mcase}(t, a_1.e_1, a_2.e_2)$. Hay dos casos:
  Si t se redujo a $\mathsf{inl}\, v$ entonces $e = e_1[a_1 := v]$ y por hipótesis de inducción existen $\tau_{e'}$ y γ_1 tales que $\tau_{e'}$ es consistente con e' y es adecuada para μ_1, $\gamma_1^\circ = \beta_1$ y $\tau_{e'}(\ell)@\gamma_1^- = e'^-$.
  Sea $\tau_e = \tau_{e'}$ y $\gamma = \mathsf{inl}\, v_1 \circ \gamma_1$, así $\gamma^\circ = \gamma_1^\circ$ y

$$\begin{array}{rcl} (\tau_e(\ell))^-@\gamma^- & = & (\tau_e(\ell)@\gamma)^- \\ & = & \mathsf{mcase}(t^-, a_1.e_1^-, a_2.e_2^-)@(v_1^-) \circ \bullet \\ & = & e_1^-[a_1 := v_1^-]@\bullet \\ & = & e_1^-[a_1 := v_1^-] = e^- \end{array}$$

  Si t se redujo a $\mathsf{inr}\, v$ entonces $e = e_2[a_2 := v]$

□

Después del siguiente lema se podrá demostrar la seguridad del sistema SM.

Lema 3.10. *Sea r un término o una expresión de* SM. *Si $\Gamma|\Delta \vdash_{\mathsf{S}} r^- : \mathsf{T}$ entonces existe un contexto de etiquetas de locaciones Σ tal que $\Gamma|\Delta|\Sigma \vdash_{\mathsf{SM}} r : \mathsf{T}$.*

Demostración. Inducción sobre $\vdash_{\mathsf{S}} r^-$.
Veamos el caso para funciones memoizadas:
A partir de $\Gamma|\Delta \vdash_{\mathsf{S}} \mathsf{mfun}(f.a.e^-) : \mathsf{T}_1 \to \mathsf{T}_2$ y por el lema de inversión se obtiene $\Gamma,\ f : \mathsf{T}_1 \to \mathsf{T}_2 \,|\, \Delta,\ a :: \mathsf{T}_1 \vdash_{\mathsf{S}} e^- : \mathsf{T}_2$.
Aplicando la hipótesis de inducción se puede obtener $\Gamma,\ f : \mathsf{T}_1 \to \mathsf{T}_2 \,|\, \Delta,\ a :: \mathsf{T}_1 | \Sigma \vdash_{\mathsf{S}} e^- : \mathsf{T}_2$ en donde Σ es el contexto de etiquetas para e^-. Así sólo es necesario agregar una etiqueta nueva a Σ que indique la locación donde se guarda la tabla para la función f

$$\Sigma' = \Sigma \cup \{\ell\} \quad \Gamma|\Delta\Sigma' \vdash_{\mathsf{SM}} \mathsf{mfun}(f.a.e^-) : \mathsf{T}_1 \to \mathsf{T}_2$$

□

3.4.1. Seguridad de SM

Ahora se puede desarrollar la prueba de seguridad de tipos para el sistema SM que se describe en el teorema 3.1.

Teorema (Seguridad de SM)**.**

1. *Si* $\Gamma| \cdot |\Sigma \vdash t : \mathsf{T}$ *y* $\mu, t \Downarrow^t v, \mu'$ *con* μ *inicial, entonces existe* Σ' *tal que* $\Gamma| \cdot |\Sigma' \vdash v : \mathsf{T}$.

2. *Si* $\Gamma| \cdot |\Sigma \vdash e : \mathsf{T}$ *y la evaluación* $\mu, e \Downarrow^e_{\beta@\ell} v, \mu'$ *se origina en una aplicación*[4] $\mu^\star, t_1 t_2 \Downarrow^t v^\star, \mu^{\star\prime}$ *con* $\mu^\star$ *inicial, entonces existe* Σ' *tal que* $\Gamma| \cdot |\Sigma' \vdash v : \mathsf{T}$.

Demostración.

- Parte (i). Supongamos $\Gamma| \cdot |\Sigma \vdash t : \mathsf{T}$ y $\mu, t \Downarrow^t v, \mu'$ con μ una memoria inicial.
  Sea $\mathcal{L}_t$ el conjunto de etiquetas que ocurren en t. Sin pérdida de generalidad se puede asumir que $dom(\mu) = \mathcal{L}_t = dom(\Sigma)$. Observemos que el tipado asegura que para cada $\ell \in dom(\tau)$ existe una única expresión e tal que $\mathsf{mfun}_\ell(f.a.e)$ ocurre en t. Por lo tanto se puede definir una tabla de funciones τ con $dom(\tau) = dom(\mu)$ al definir $\tau(\ell) = \mathsf{mfun}(f.a.e^-)$ para cada $\ell \in dom(\tau)$. De esta forma τ es consistente con t por construcción y es compatible con μ dada la inicialidad de μ. Luego entonces se puede aplicar la parte (i) de la proposición 3.2 para obtener a $\tau' \supseteq \tau$ tal que τ' es consistente con v, compatible con μ' y $t^- \Downarrow^t_p v^-$.
  Por otro lado, de la proposición 3.1 se tiene la derivación de tipo para t^-, $\Gamma|\cdot \vdash t^- : \mathsf{T}$, la cual junto con $t^- \Downarrow^t_p v^-$ implica, por la seguridad del sistema S (prop. 3.2), que $\Gamma|\cdot \vdash v^- : \mathsf{T}$.
  Finalmente el lema 3.10 produce Σ' tal que $\Gamma| \cdot |\Sigma' \vdash v : \mathsf{T}$.

[4]Es decir que $\mu, e \Downarrow^e_{\beta@\ell} v\, \mu'$ se obtiene como parte de la derivación de la premisa que involucra a $\Downarrow^e_{\bullet@\ell'}$ en la regla (Emapply).

- Parte (ii). Ahora supongamos $\Gamma|\cdot|\Sigma \vdash e : \mathsf{T}$ y $\mu, e \Downarrow^{e}_{\beta@\ell} v, \mu'$ originada en una aplicación $\mu^\star, t_1 t_2 \Downarrow^{t}_{p} v^\star, \mu^{\star\prime}$ con $\mu^\star$ inicial. Por el lema 3.9 existen τ_e consistente con e y compatible con μ, y γ tal que $\tau_e(\ell)@\gamma^- = e^-$ y $\gamma^\circ = \beta$.
  A partir de la proposición 3.1 se puede obtener la derivación $\Gamma|\cdot \vdash e^- : \mathsf{T}$ y por la seguridad del sistema S, tomando la evaluación $e^- \Downarrow^{e}_{\beta;v_f} v^-$, se obtiene $\Gamma|\cdot \vdash v^- : \mathsf{T}$. Finalmente se agrega el contexto de etiquetas Σ' de acuerdo al lema 3.10 $\Gamma|\cdot|\Sigma' \vdash v : \mathsf{T}$.

□

Se ha demostrado el objetivo principal de este capítulo. Sin embargo, a pesar de que el sistema S no tiene efectos laterales explícitos, desde un punto de vista estricto, este sistema no es puro, se conserva la distinción entre términos y expresiones así como la selectividad. Para resolver este problema se da una traducción de S al lenguaje funcional puro PCF. Así, se reduce indirectamente el sistema para memoización selectiva a uno puramente funcional demostrando que la semántica original que incluye memoización es correcta respecto a la semántica pura de PCF.

3.5. Traducción a PCF

La traducción anterior, a pesar de eliminar efectos laterales, no queda del todo pura, el sistema S sólo es un auxiliar para probar la seguridad del sistema SM el cual maneja funciones memoizadas de manera selectiva. La impureza en la semántica recae en el uso de tablas de funciones y la selectividad para analizar los argumentos de funciones memoizadas, así como la conservación del constructor return.
En [2] se demuestra que la evaluación de cualquier función memoizada devolvería el mismo valor si no estuviera memoizada mediante la traducción del sistema SM a uno que no incluya memoización. Esta idea no queda totalmente expuesta y es la que se interpreta como seguridad del sistema **MFL** de Acar et al. Un sistema puro debe de eliminar completamente los efectos de la memoización. En esta sección se describe la traducción al sistema **PCF** (Programming Computable Functions) de D. Scott y G. Plotkin, un sistema puro y bien conocido en el ámbito de la programación funcional.

La traducción en tipos es la función identidad excepto para el caso del tipo box: $(\Box\,\mathsf{T})^* = \mathsf{T}^*$. Ya que el tipo $\Box\,\mathsf{T}$ indicaba cuándo un argumento de una función memoizada debía ser explorado para registrar las dependencias entre él y el valor devuelto por la función. La traducción de una función memoizada es el operador de punto fijo, comúnmente llamado fix para funciones recursivas; en este sistema sólo lo llamaremos fun.
Los términos y expresiones se colapsan en términos de **PCF**, obsérvese que las variables de recurso a son mapeadas a variables x_a de **PCF**.

r	r^*
x	x
a	x_a
$o(t, \ldots, t)$	$o(t^*, \ldots, t^*)$
$\star$	$\star$
n	n
$\lambda x : \mathsf{T}\,.r$	$\lambda x : \mathsf{T}\,.r^*$
$\mathsf{mfun}(f.a.e)$	$\mathsf{fun}(f.x_a.e^*)$
rs	r^*s^*
$\mathsf{inl}_{\mathsf{T}}\, r$	$\mathsf{inl}_{\mathsf{T}}\, r^*$
$\mathsf{inr}_{\mathsf{T}}\, s$	$\mathsf{inr}_{\mathsf{T}}\, s^*$
$\mathsf{case}(r, x.s, y.t)$	$\mathsf{case}(r^*, x.s^*, y.t^*)$
$\langle r, s\rangle$	$\langle r^*, s^*\rangle$
$\mathsf{fst}\, r$	$\mathsf{fst}\, r^*$
$\mathsf{snd}\, r$	$\mathsf{snd}\, r^*$
$\mathsf{box}\, t$	t^*
$\mathsf{return}\, t$	t^*
$\mathsf{let\,box}(t,\ x.e)$	$\mathsf{let}(t^*,\ x.e^*)$
$\mathsf{letprod}(t,\ a_1.a_2.e)$	$\mathsf{let}(\mathsf{snd}\, t^*,\ x_{a_2}.(\mathsf{let}(\mathsf{fst}\, t^*,\ x_{a_1}.e^*)))$
$\mathsf{mcase}(t,\ a_1.e_1,\ a_2.e_2)$	$\mathsf{case}(t^*,\ x_{a_1}.e_1^*,\ x_{a_2}.e_2^*)$

Las funciones con nombre fun y las expresiones let se consideran primitivas de **PCF** a pesar de que son azúcar sintáctica.

Los siguientes resultados prueban que la traducción $(\cdot)^*$ es fiel.

Proposición 3.3. *Sea r un término o una expresión. Si $\Gamma|\Delta \vdash r : \mathsf{T}$ entonces $\Gamma^*,\ \Delta^* \vdash r^* : \mathsf{T}^*$ donde $\Gamma^* = \{x : \mathsf{T}^* \,|x : \mathsf{T} \in \Gamma\}$ y $\Delta^* = \{x_a|a :: \mathsf{T} \in \Delta\}$.*

Demostración.

Inducción sobre la estructura de r:

- $r = \mathsf{box}\, t$ y $\Gamma \,|\, \Delta \vdash \mathsf{box}\, t : \Box\, \mathsf{T}$.
  Utilizando el lema de inversión se obtiene $\Gamma \,|\, \cdot \vdash t : \mathsf{T}$ y aplicándole la hipótesis de inducción obtenemos la derivación del tipo $\Gamma^* \vdash t^* : \mathsf{T}^*$. De ésta se puede obtener que $\Gamma^*,\ \Delta \vdash t^* : \mathsf{T}^*$ por la propiedad de monotonía, pero observemos que $(\mathsf{box}\, t)^* = t^*$ y $(\Box\, \mathsf{T})^* = \mathsf{T}^*$, por lo tanto $\Gamma^*,\ \Delta \vdash (\mathsf{box}\, t)^* : (\Box\, \mathsf{T})^*$.

- $r = \mathsf{mfun}(f.a.e)$ y $\Gamma \,|\, \Delta \vdash \mathsf{mfun}(f.a.e) : \mathsf{T}_1 \to \mathsf{T}_2$.
  Por el lema de inversión se obtiene $\Gamma,\ f : \mathsf{T}_1 \to \mathsf{T}_2 \,|\, \Delta,\ a :: \mathsf{T}_1 \vdash e : \mathsf{T}_2$ y aplicando la hipótesis de inducción: $\Gamma^*,\ f : \mathsf{T}_1{}^* \to \mathsf{T}_2{}^*,\ \Delta^*,\ x_a :: \mathsf{T}_1{}^* \vdash e^* : \mathsf{T}_2{}^*$. Con este resultado se puede derivar el tipo de $r^* = \mathsf{fun}(f.x_a.e^*)$: $\Gamma^*, \Delta^* \vdash \mathsf{fun}(f.x_a.e^*) : \mathsf{T}_1{}^* \to \mathsf{T}_2{}^*$.

- $r = \mathsf{return}\, t$ y $\Gamma \mid \Delta \vdash \mathsf{return}\, t : \mathsf{T}$.
  Sabemos que $(\mathsf{return}\, t)^* = t^*$ y por el lema de inversión obtenemos $\Gamma \mid \cdot \vdash t : \mathsf{T}$. Aplicando la hipótesis de inducción se obtiene $\Gamma^* \vdash t^* : \mathsf{T}^*$ y por la propiedad de monotonía se puede agregar el contexto faltante Γ^*, $\Delta \vdash t^* : \mathsf{T}^*$.

- $r = \mathsf{let\,box}(t,\, x.e)$ y $\Gamma \mid \Delta \vdash \mathsf{let\,box}(t,\, x.e) : \mathsf{T}$.
  Por el lema de inversión se obtienen dos derivaciones de tipo $\Gamma \mid \Delta \vdash t : \Box\, \mathsf{T}_1$ y $\Gamma, x : \mathsf{T}_1 \mid \Delta \vdash e : \mathsf{T}_2$. Al aplicar la hipótesis de inducción a ambas se obtienen las hipótesis para derivar el tipo de $r^* = \mathsf{let}(t^*,\, x.e^*) = (\mathsf{let}(t,\, x.e))^*$:
  De Γ^*, $\Delta^* \vdash t^* : \mathsf{T}_1{}^*$ y $\Gamma^*, x : \mathsf{T}_1{}^*$, $\Delta^* \vdash e^* : \mathsf{T}_2{}^*$ se deriva Γ^*, $\Delta^* \vdash \mathsf{let}(t^*,\, x.e^*) : \mathsf{T}_1{}^*$

□

Proposición 3.4. *La traducción* $(\cdot)^*$ *satisface las siguientes:*

1. *Si* $t \Downarrow_p^t v$ *entonces* $t^* \Downarrow_{\mathsf{PCF}} v^*$.
2. *Si* $e \Downarrow_{\beta;v_f}^e v$ *entonces* $e^* \Downarrow_{\mathsf{PCF}} v^*$.

Demostración.
Las dos partes se demuestran simultáneamente por inducción sobre las relaciones de evaluación $\Downarrow_p^t$ y $\Downarrow_{\beta;v_f}^e$, se muestran algunos de los casos más relevantes:

- $t = t_1\, t_2$ y $t_1\, t_2 \Downarrow_p^t v$ derivada de $t_1 \Downarrow_p^t v_1 = \mathsf{mfun}(f.a.e)$, $t_2 \Downarrow_p^t v_2$ y $e[f, a := v_1,\, v_2] \Downarrow_{\bullet;\, v_1}^e v$.
  A cada una de las anteriores se les aplica la hipótesis de inducción correspondiente: ${t_1}^* \Downarrow_{\mathsf{PCF}} {v_1}^* = \mathsf{fun}(f.x_a.e^*)$, ${t_2}^* \Downarrow_{\mathsf{PCF}} {v_2}^*$ y $e^*[f, x_a := {v_1}^*,\, {v_2}^*] \Downarrow_{\mathsf{PCF}} v^*$ para obtener la evaluación $(t_1\, t_2)^* \Downarrow_{\mathsf{PCF}} v^*$.

- $e = \mathsf{return}\ t$ y $\mathsf{return}\ t \Downarrow_{\beta;v_f}^e v$ derivada de $v_f @ \beta = \mathsf{return}\ t$ y $t \Downarrow_p^t v$.
  Calculando e^* se obtiene $(\mathsf{return}\ t)^* = t^*$, basta con aplicar la hipótesis de inducción a la evaluación $t \Downarrow_p^t v$ para encontrar la evaluación de $(\mathsf{return}\ t)^*$: $(\mathsf{return}\ t)^* \Downarrow_{\mathsf{PCF}} v^*$

- $e = \mathsf{mcase}(t,\, a_1.e_1,\, a_2.e_2)$ y $\mathsf{mcase}\,(t,\, a_1.e_1,\, a_2.e_2) \Downarrow_{\beta;v_f}^e, v_2$ derivada de $t \Downarrow_p^t \mathsf{inr}\, v$ y $e_2[a_2 := v] \Downarrow_{\mathsf{inr}\cdot\beta;v_f}^e v_2$.
  Aplicando el caso de la hipótesis de inducción correspondiente se obtienen:
  $t^* \Downarrow_{\mathsf{PCF}} \mathsf{inr}\, v^*$ y ${e_2}^*[x_{a_2} := v^*] \Downarrow_{\mathsf{PCF}} {v_2}^*$, las cuales derivan la evaluación de
  $e^* = \mathsf{case}(t^*,\, x_{a_1}.e_1^*,\, x_{a_2}.e_2^*)$

□

Finalmente se demostrado que la traducción del sistema SM a PCF pasando por el sistema S es fiel. Además la propiedad de transparencia referncial se conserva ya que el resultado de una función memoizada es el mismo que se obtendría si no se memoizara.

Conclusiones y Trabajo Futuro

Si bien, la técnica de memoización es una técnica para la optimización de programas puede suceder que no sea efectiva. A lo largo de este trabajo se revisaron diferentes enfoques para implementar la memoización, algunos menos intuitivos o menos efectivos. Todos ellos fueron desarrollados dentro del ámbito funcional puro, algunos llevados a la implementación en el lenguaje Haskell donde se pudo comparar su desempeño a través del número de reducciones hechas al momento de la evaluación. Los desempeños dependen de la forma de implementación: de las estructuras utilizadas para el manejo de la tabla de memoización y del manejo de las funciones memoizadas.
Los enfoques estudiados revisan diversas formas y técnicas para manejar e implementar la técnica de memoización:

- Para indexar la tabla se revisaron y estudiaron diversas formas de indexación. Desde la más común en donde se toman los argumentos de las funciones memoizadas como llaves, hasta la que requiere de una sintaxis especial para identificar las dependencias generadas por los argumentos e indexar la tabla utilizando el recuento de estas dependencias.

- Para la implementación y manejo de las tablas de memoización se compararon diferentes estructuras. La estructura más sencilla es una lista que almacena tablas las cuales son parejas que incluyen el nombre de la función memoizada y la tabla representada por una lista de parejas argumento-resultado. Otra estructura usada, que en escencia continúa siendo una lista, es la mónada de estado. Con el polimorfismo politípico de segundo orden se definieron las funciones de manejo de una tabla de memoización como funciones inversas y finalmente se simularon la memoria y las tablas mediante funciones parciales en la parte de memoización selectiva.

Para transformar una función recursiva a una función memoizada se revisaron los enfoques monádico, politípico y selectivo. Del primer enfoque se revisaron dos transformaciones: la primera propuesta por Frost, utiliza la función `memoize` la cual recibe como argumento una función que incluye la mónada de estado. Este argumento debe ser transformado antes de ser programado, es decir a partir de una solución recursiva al problema, ésta se modifica para incluir la mónada de estado y así manejar la tabla de memoización. La segunda propuesta por Brown y Cook, describe el proceso *monadification* el cual es más laborioso que el anterior. La transformación de la solución recursiva incluye el uso de mónadas pero para simular la programación imperativa además aprovecha la ventaja de la evaluación perezosa y manejar la tabla de memoización a través de un diccionario.

El segundo enfoque aplica el polimorfismo politípico de segundo orden usado para definir la tabla de memoización a la implementación recursiva. En este enfoque no es necesaria una transformación previa de la solución recursiva, de hecho es más elegante el uso de la tabla de memoización al verla como la aplicación de dos funciones, primero la que agrega valores a la tabla seguida de la función que accesa a la tabla para buscar un resultado almacenado.

La programación funcional pura ayudó a la implementación sencilla de los enfoques utilizados. Gracias a las ventajas que provee se pudo simular la programación no funcional y sacar provecho de las definiciones de tipos de datos para implementar las tablas. Haskell permitió comparar los enfoques de las implementaciones estudiadas. La inclusión de la programación genérica dentro de la programación funcional pura permitió hacer uso del concepto de herencia, característica importante de los lenguajes orientados a objetos, para definir atributos y parámetros dejando de lado la implementación particular de los métodos a la clase que hereda la clase mixin (pág. 21).

El tercero y último define un trato especial de las funciones a memoizar, tanto sintáctica como semánticamente. En este enfoque, el selectivo, se desarrolló el sistema SM semejante al sistema MFL de [2] pero con las siguientes mejoras desde un punto de vista teórico:

- cambio de la modalidad $!\mathsf{T}$ (*bang*) por el tipo modal $\Box\mathsf{T}$ correspondiente a la modalidad de necesidad expuesta en [26]
- la semántica estática provee un seguimiento exacto de las etiquetas que figuran dentro de un término o expresión
- del punto anterior se asegura estáticamente la unicidad de una etiqueta para las funciones memoizadas
- la seguridad del sistema SM se demuestra mediante una traducción a un sistema sin efectos
- finalmente se traduce al sistema puramente funcional **PCF**

Este último punto muestra que la memoización selectiva no afecta el resultado final de una evaluación en donde esté involucrada una función memoizada, es decir se conserva la transparencia referencial y se obtiene una optimización efectiva.

Dentro del trabajo futuro se busca realizar una implementación en Haskell basada en la memoización politípica revisada en el capítulo anterior y desarrollada en [13]. También se busca formular un sistema que, bajo la correspondencia Curry-Howard, corresponda al fragmento de necesidad de la reconstrucción con juicios de la lógica modal vista en [26]. En particular se debe marcar una diferencia entre las reglas de tipado de términos y de expresiones a pesar de que se encuentran definidas mutuamente para su evaluación. Además el constructor return debe desaparecer para tener una correspondencia completa. Algunas posibles extensiones son:

1. agregar el tipo $\bigcirc\,\mathsf{T}$ para representar la memoización monádica y otros cómputos de tipo T.

2. incluir tipos compuestos pero las dependencias de dato deben ser refinadas

3. incluir tipos inductivos como aparece en [18] para modelar el principio de recursión por curso de valores

El esquema de recursión por curso de valores generaliza la iteración, veamos la definición de este principio para los números naturales:

Definición 3.11. *Sean* nat *el tipo de dato para los números naturales,* C *un (co) tipo de datos,* $c_0, \ldots, c_k \in C$ *y la función* $g : \mathsf{Colist}C \to C$[5]*. La función* $f : \mathsf{nat} \to C$ *se define por curso de valores con casos base* $c_0, \ldots, c_k$ *y función de paso* g *si y sólo si:*

- $f(0) = c_0,\ f(1) = c_1, \ldots,\ f(k) = c_k$
- $f(n+1) = g(\mathsf{rcd}fn),\ n \geq k$

donde $\mathsf{rcd} : (\mathsf{nat} \to C) \to \mathsf{nat} \to \mathsf{Colist}(C)$ *es la función que registra la historia de* f*:* $\mathsf{rcd}fn = [fn, f(n-1), \ldots, f0]$

Una función interesante definida utilizando este principio es la que genera los números de Catalán:

$$C_0 = 1, \qquad C_{n+1} = C_0C_n + C_1C_{n-1} + \ldots C_{n-1}C_1 + C_nC_0$$

Se puede extender el sistema de tipos al agregar μF para modelar las álgebras iniciales que permiten definir tipos inductivos como los naturales, las listas finitas de objetos, etc. Así los tipos básicos ya no se considerarían básicos sino una construcción permitida por la gramática como lo muestra la siguiente regla:

$$\frac{\Gamma \vdash t : F(\mu F)}{\Gamma \vdash t : \mu F}\ (\mu I)$$

[5] Colist es el tipo de dato de las listas no vacías y posiblemente infinitas definidas por el funtor $FX = C \times (1 + X)$

Bibliografía

[1] Acar U. A., G. E. Blelloch and R. Harper, *Selective memoization*, Technical Report CMU-CS-02-194, Carnegie Mellon University, Computer Science Department, 2002.

[2] Acar U. A., G. E. Blelloch and R. Harper, *Selective memoization*, In Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2003), 14-25.

[3] Bird, R., "Introducción a la Programación Funcional con Haskell", Prentice-Hall, 2000.

[4] Brown, D. and W. R. Cook, *Monadic Memoization Mixins*, 2006.

[5] Cormen T. H., C. Leiserson and R. Rivest, "Introduction to algorithms", MIT Press, 1990.

[6] Fairtlough, M. and M. Mendler, *Propositional Lax Logic*, Information and Computation **137** (1997), 1-33.

[7] Field, A., P. Harrison, *Functional Programming*, International computer science series, Addison-Wesley, 1988.

[8] Frost, R., *Monadic Memoization towards Correctness-Preserving Reduction of Search*, Lecture Notes in Artificial Intelligence **2671** (2003), 66-80.

[9] González-Huesca, L., *Coinducción: de la Teoría de Categorías a la Programación Funcional*, Tesis de Licenciatura, UNAM, 2007.

[10] Heydon A., R. Levin and Y. Yu, *Caching function calls using precise dependencies*, In Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation, 311-320.

[11] Hinze R., *A New Approach to Generic Functional Programming*, In Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (2000), 119-132.

[12] Hinze R., *Generalizing Generalized Tries*, 1998.

[13] Hinze R., *Memo functions, polytypically!*, Proceedings of the 2nd Workshop on Generic Programming (2000), 17-32.

[14] Hinze R. and J. Jeuring, *Generic Haskell: practice and theory*, In Generic Programming, Advanced Lectures **2793** (2003) of LNCS, 1-56.

[15] Jeuring J. and P. Jansson, *Polytypic Programming*, 2nd Int. School on Advanced Functional Programming (1996), 68-114, Springer-Verlag.

[16] Martí, N., Y. Ortega y J.A. Verdejo, "Estructuras de datos y métodos algorítmicos", Prentice Hall, Madrid, 2004.

[17] Michie D., *"Memo"Functions and Machine Learning*, Nature (1968), 218:19-22.

[18] Miranda-Perea F. E., *Some Remarks on Type Systems for Course-of-value Recursion* Electronic Notes in Theoretical Computer Science **247** (2009), 103-121.

[19] Miranda-Perea, F. E. and L. González-Huesca, *Selective Memoization with Box Types*, Electronic Notes in Theoretical Computer Science **256** (2009), 67-85.

[20] Miyamoto K. and A. Igarashi, *A modal foundation for secure information flow*, In Proceedings of Workshop on Foundations of Computer Security (2004), 187-203.

[21] Moggi E., *Computational lambda-calculus and monads*, In Proceedings of the Fourth Annual Symposium on Logic in computer science (1989), 14-23.

[22] Moody J., *Logical Mobility and Locality Types*, In Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (2004), 69-84.

[23] Norvig P., *Techniques for Automatic Memoization with Applications to Context-Free Parsing*, Computational Linguistics **17** (1991), 91-98.

[24] Okasaki C. and A. Gill, *Fast Mergeable Integer Maps*, In Workshop on ML (1998), 77-86.

[25] Okasaki, C., "Purely Functional Data Structures", Cambridge University Press, 1998.

[26] Pfenning F. and R. Davies, *A judgmental reconstruction of modal logic*, Mathematical Structures in Computer Science **11** (2001), 511-540.

[27] Pierce B. C., "Types and Programming Languages", MIT Press, 2002.

[28] Pugh W. and T. Teitelbaum, *Incremental Computation via Function Caching*, In Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (1989), 315-328.

[29] Russell, S. and P. Norvig, "Artificial Intelligence, A modern approach", Segunda edición, Prentice Hall, 2003.

[30] Scott, D.S., "Relating Theories of the Lambda-Calculus", In R. Hindley and J. Selding, editors, *To H.B. Curry: essays in Combinatory Logic, lambda calculus and Formalisms.*, Academic Press, 1980.

[31] Thompson, S., "Haskell: The craft of funtional programming", Segunda Edición, Pearson Addison-Wesley, 1999.

[32] van Oosten, J., *Basic Category Theory*, Brics, 1995.

[33] Vardi, M.Y., "Why is modal logic so robustly decidable", In N. Immerman, P. Kolaitis (Eds.), Descriptive Complexity and Finite Models, In Discrete Mathematics and Theoretical Computer Science **31** DIMACS (1997), 149-184.

[34] Yuse Y. and A. Igarashi, *A modal type system for multi-level generating extensions with persistent code*, In Proceedings of the 8th ACM SIGPLAN international conference on Principles and practice of declarative programming (2006), 201-212.

[35] AllExperts, "memoization", in AllExperts Free Encyclopedia, http://en.allexperts.com/e/m/me/memoization.htm, accessed December 2008-October 2009.

[36] AllExperts, "referential transparency", in AllExperts Free Encyclopedia, http://en.allexperts.com/e/r/re/referential_transparency.htm, accessed March-October 2009.

[37] AllExperts, "lazy evaluation", in AllExperts Free Encyclopedia, http://en.allexperts.com/e/l/la/lazy_evaluation.htm, accessed May-October 2009.

[38] Black, Paul E., "memoization", in Dictionary of Algorithms and Data Structures [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology, (14 August 2008) http://www.itl.nist.gov/div897/sqg/dads/HTML/memoize.html, accessed January 2009.

[39] Wikipedia, "Dynamic programming", in Wikipedia The Free Encyclopedia [online], (2009), http://en.wikipedia.org/w/index.php?title=Dynamic_programming&oldid=326087498, accessed March 2009-October 2009.

[40] Wikipedia, "Event calculus", in Wikipedia The Free Encyclopedia [online], (2009), http://en.wikipedia.org/w/index.php?title=Event_calculus&oldid=329776556, accessed January 2010.

[41] Wikipedia, "Formal system", in Wikipedia The Free Encyclopedia[online], (2010), http://en.wikipedia.org/w/index.php?title=Formal_system&oldid=337256819, accessed January 2010.

[42] Wikipedia, "Functional programming", in Wikipedia The Free Encyclopedia [online], (2009), http://en.wikipedia.org/w/index.php?title=Functional_programming&oldid=326530835, accessed May 2009-October 2009.

[43] Wikipedia, "Logic", in Wikipedia The Free Encyclopedia [online], (2010), http://en.wikipedia.org/w/index.php?title=Logic&oldid=338552386, accessed January 2010.

[44] Wikipedia, "Modal logic", in Wikipedia The Free Encyclopedia [online], (2009), http://en.wikipedia.org/w/index.php?title=Modal_logic&oldid=325132991, accessed March 2009-October 2009.

[45] HaskellWiki, http://www.haskell.org/haskellwiki/, accessed October 2008-October 2009.