Universidad ORT Uruguay
Facultad de Ingeniería

Puerto de Shen para Erlang

Entregado como requisito para la obtención del título de
Ingeniero de Sistemas

Sebastián Borrazás - 166089
Tutor: Álvaro Tasistro

2018

Declaración de autoría

Yo, Sebastián Borrazás, declaro que el trabajo que se presenta en esta obra es de mi propia mano. Aseguro que:

- La obra fue producida en su totalidad mientras realizaba la tesis;
- Cuando he consultado trabajos por otros, lo he atribuido con claridad;
- Cuando he citado obras de otros autores, he indicado las fuentes. Con excepción de dichas citas, la obra es enteramente mía;
- En la obra, he acusado recibo las ayudas recibidas;
- Cuando la obra se basa en trabajo realizado conjuntamente con otros, he explicado claramente qué parte fue contribuida por dichos terceros y qué parte fue contribuida por mi;
- Ninguna parte de este trabajo ha sido publicada previamente a su entrega, excepto los casos en que se han realizado aclaraciones correspondiente.

Sebastián Borrazás

22/08/2018

Agradecimientos

Deseo agradecer a mi familia por el apoyo y la motivación para avanzar con mis estudios.

Agradezco también a Álvaro Tasistro por ayudarme a realizar este trabajo de tesis de gran interés para mi.

Abstract

La evolución de las herramientas para la construcción de compiladores e intérpretes posibilita ahora desarrollarlos de forma fácil. Este motivo, sumado a la mejora en la capacidad de procesamiento de las computadoras, permitió el desarrollo de varios nuevos lenguajes, tanto de propósito general como de dominio específico, que ahora se pueden utilizar en ambientes de producción para aplicaciones de uso práctico.

La creación de nuevos lenguajes trae consigo nuevos problemas, fundamentalmente de portabilidad, donde los programas escritos en estos lenguajes deben interactuar con programas, lenguajes, sistemas operativos y arquitecturas prexistentes.

Para solucionar este problema, muchas veces se definen nuevos lenguajes intermedios con un número reducido de primitivas y sintaxis abreviadas (también llamados *kernels*) para facilitar la compilación y lograr "portar" el lenguaje original a distintos otros lenguajes y plataformas. Esta forma de "portar" el lenguaje utilizando un lenguaje intermedio reducido se hace a través de la construcción de "puertos". Un "puerto" ejecuta el código intermedio sobre una plataforma particular, por lo que depende de esta plataforma, pero es fácilmente construido debido a la simplicidad del lenguaje intermedio. Cada puerto que se crea para ejecutar el lenguaje intermedio extiende la portabilidad del lenguaje original.

El propósito de este trabajo es mostrar el proceso de construcción de un puerto de un nuevo lenguaje llamado Shen, diseñado para ser fácilmente portable a través de un lenguaje intermedio llamado Kλ utilizando Erlang como el lenguaje al que se "porta".

Todos los conceptos, técnicas, decisiones tomadas y pasos necesarios para la construcción del puerto están fundamentados utilizando el modelo computacional de una formalización de Kλ y Erlang. Esta formalización es la contribución principal de este trabajo.

Palabras clave

shen, puerto, erlang, klambda, compilador, intérprete

Índice general

Capítulo 1

Introducción

Esta introducción plantea la motivación detrás de la construcción del puerto de Shen para Erlang y describe el marco de trabajo que utilizaremos.

1.1. Motivación

Al momento de definir un lenguaje es un requerimiento deseado que sus programas se puedan ejecutar en distintas plataformas. Por este motivo, algunos lenguajes eligen compilar programas a un lenguaje intermedio (llamado *kernel*) con un número reducido de primitivas para luego ser compilados nuevamente a código que sí depende de la plataforma. Esta técnica se realiza a través de la creación de "puertos", que son dependientes de la plataforma pero fácilmente implementables.

La motivación principal de este trabajo es formalizar la semántica operacional de los lenguajes involucrados en la construcción de un puerto para probar su corrección.

1.1.1. ¿Qué es un "puerto"?

En primer lugar, definimos el concepto de *"puerto"*, que actualmente se utiliza en la industria con muchas variantes cuyo uso depende del contexto. En este trabajo, utilizaremos este término exclusivamente en el contexto del software.

Incluso dentro del contexto del software, existen varios significados para el término "puerto". Algunos ejemplos son:

- Puerto de red: utilizado como interfaz que se expone a sistemas externos para enviar y recibir mensajes.
- Puerto de videojuego: en el que se adapta un videojuego para ser ejecutado sobre otra consola de videojuego distinta a la que construyó el juego original.
- Puerto de lenguajes de programación: en el que se traduce o ejecuta un lenguaje de programación para ser portado en un ambiente distinto de aquél para el cual originalmente fue creado.

Este último uso de "puerto" es el que utilizaremos en este trabajo. Más específicamente, llamaremos "puerto" a todo programa que permite ejecutar el lenguaje de programación original en un nuevo contexto de ejecución. Este programa puede ser un compilador – utilizado para traducir del lenguaje original a otro – o un intérprete – que ejecuta directamente el código del lenguaje original.

El beneficio que provee un puerto de este tipo es poder independizar al compilador del lenguaje original de la plataforma sobre la que debe ejecutar el programa a través de la generación de código intermedio. El código intermedio define una máquina virtual, también llamada máquina virtual de aplicación [1], que determina cómo se deberá ejecutar el código generado, es decir, define su semántica operacional. Esto implica que deberá haber una trazabilidad de la semántica operacional del código original y la semántica de la máquina virtual.

El puerto encargado de ejecutar (o interpretar) el lenguaje intermedio sí depende de la plataforma donde deberá ser ejecutado, pero es independiente del lenguaje original debido a que la compilación del lenguaje original al intermedio ya está por lo general implementada. Este beneficio es una característica muchas veces deseada en los sistemas de software, y se llama "portabilidad". Cuanto mayor la portabilidad de un sistema, mejor es [2]. Un sistema portable no sólo tiene mayor independencia de la plataforma sobre la que se ejecuta, sino que además permite la utilización de programas en distintas tecnologías interactuar entre ellos. Esto se define como la "virtualización de software" [1].

En la figura 1.1 se muestra un diagrama de los elementos que interactúan con el puerto y el flujo de código consumido y generado por los distintos

elementos. Los círculos representan compiladores o intérpretes que toman el código de entrada y generan el código de salida o simplemente ejecutan.

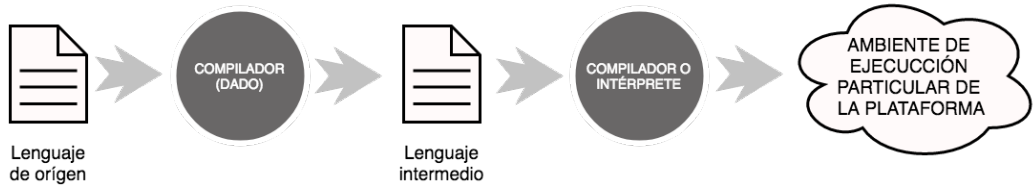

Figura 1.1: Estructura de un puerto

En la figura se muestran los siguientes elementos:

1. El código del lenguaje de origen: tiene la intención de ser útil y legible.
2. El compilador: toma el código del lenguaje de origen y genera el código del lenguaje intermedio. No depende de la plataforma y será para todos los puertos el mismo.
3. El código del lenguaje intermedio: está determinado por un número reducido de primitivas y es por lo general simple y fácil de compilar o interpretar.
4. El compilador o intérprete: toma el código del lenguaje intermedio y genera código o ejecuta un intérprete dependiente de la plataforma. Éste deberá ser construido para cada plataforma donde se desee ejecutar el código escrito en el lenguaje original.
5. La plataforma: el ambiente donde ejecutará el código compilado o intérprete del puerto.

Este flujo del código compilado puede repetirse varias veces; en estos casos el código de salida del compilador del paso 3 genera nuevamente código intermedio de otro lenguaje que deberá ser nuevamente interpretado o compilado. A efectos prácticos, esto se repite hasta generar código de máquina que será ejecutado por el hardware. En algunos casos, sin embargo, se utiliza el lenguaje intermedio para definir la semántica operacional del lenguaje original debido a que el lenguaje intermedio es generalmente más reducido (con menos primitivas) que el lenguaje original y por lo tanto más fácil de definir. Esta última forma de definir un lenguaje a través de otro mediante la compilación del original al intermedio junto la semántica del lenguaje intermedio es una forma de definir la semántica operacional de un lenguaje llamado "lenguaje *kernel*" [1].

1.2. Propuesta de trabajo

Para lograr construir un puerto de Shen para Erlang correctamente necesitaremos definir la semántica operacional de los lenguajes sobre los que trabajaremos para entender cómo compilar de uno a otro y luego probar si este proceso de compilación es correcto.

La semántica de un lenguaje define qué hace un programa al ejecutarse. Existen varias formas de definirla, pero en este trabajo mencionaremos dos formas que se utilizan en distintas partes del flujo de compilación del puerto:

- A través de una formalización utilizando reglas de inferencia dentro de un modelo computacional que determinan la evaluación o ejecución para cada una de las expresiones o instrucciones del lenguaje.
- A través de un lenguaje intermedio (llamado "lenguaje *kernel*") que define el comportamiento de un programa en base a su traducción al lenguaje *kernel*. Este lenguaje *kernel* generalmente es un lenguaje simple, que contiene un número reducido de primitivas, y por lo tanto fácilmente definible a través de una formalización.

Ambas formas de definir la semántica de un lenguaje permiten razonar formalmente y de forma abstracta sobre lo que debe realizar el compilador o intérprete al ejecutar programas del lenguaje, sin importar las herramientas utilizadas para su implementación o el *hardware* sobre el que deberá correr.

Existe también una forma informal de definir la semántica de un lenguaje mediante la utilización del lenguaje natural junto con ejemplos para describir cómo debe ejecutar un programa. Algunas de las tecnologías utilizadas en este trabajo están definidas de esta forma, pero para lograr un adecuado análisis del puerto formalizaremos su ejecución a través de las semánticas operacionales.

El orden en el que definiremos estas formalizaciones es el siguiente:

1. Definir la semántica operacional del lenguaje intermedio del puerto a desarrollar (Kλ).
2. Definir la semántica operacional del lenguaje de destino del puerto (Erlang).

3. Definir funciones y estructuras para realizar la compilación del lenguaje intermedio al lenguaje de destino.

4. Definir una formalización para determinar la corrección del compilador a implementar y luego demostrarlo.

1.3. Tecnologías

El proceso de compilación del puerto a construir en este trabajo involucra distintas tecnologías y lenguajes que deberán ser procesados de alguna forma para finalmente lograr ser ejecutados sobre alguna plataforma. Para este trabajo se decidió utilizar Kλ como lenguaje intermedio que deberá ser finalmente compilado a BEAM (la máquina virtual de Erlang), utilizando Erlang como herramienta.

1.3.1. ¿Por qué Shen?

Shen es un lenguaje reciente que incorpora una serie de características que lo hacen muy particular. Fundamentalmente es un lenguaje opcionalmente tipado con chequeo estático basado en el cálculo de secuentes, pero también provee varias facilidades para programar. Estas características las veremos más adelante en detalle.

A su vez, Shen se creó con la intención de ser portable a través de Kλ, un lenguaje más reducido y simple al que Shen compila. La compilación de Shen a Kλ ya está dada por las propia librería estándar de Shen. La intención del autor de Shen (Mark Tarver) es lograr que distintos desarrolladores implementen un puerto de Shen a través de la compilación de Kλ para ejecutar programas en Shen en distintos ambientes de ejecución y así lograr interoperabilidad con otras tecnologías.

1.3.2. ¿Por qué Erlang?

Al igual que Kλ, Erlang es un lenguaje funcional simple (con pocas primitivas) y por lo tanto es fácil compilar a él. Además, al igual que Shen, Erlang compila a un lenguaje de máquina virtual, BEAM, que luego es ejecutado por la máquina virtual de BEAM para su ejecución. Se puede decir entonces

que tanto Shen como Erlang son portables, y esta portabilidad se logra a través de la utilización de puertos.

Adicionalmente, a través de la creación del puerto de Shen para Erlang estamos mejorando la portabilidad de Shen y por lo tanto a Shen como lenguaje.

1.4. Desarrollo del trabajo

A continuación se menciona una breve descripción de los capítulos a desarrollar:

- En el capítulo 2 describimos el lenguaje Shen, su origen, su sintaxis, semántica y su proceso de compilación a Kλ, el kernel sobre el cual Shen está construido y al cual Shen compila para ser ejecutado.
- En el capítulo 3 definimos de forma detallada la sintaxis y semántica de Kλ, el lenguaje que deberá ser soportado por el puerto a desarrollar.
- En el capítulo 4 describimos Erlang, el lenguaje utilizado para compilar al lenguaje de la VM de BEAM que será utilizado para construir el puerto. Describimos también su relación con BEAM.
- En el capítulo 5 desarrollamos el proceso de construcción del puerto, utilizando los lenguajes antes mencionados para la construcción de un compilador de Kλ a Erlang (y finalmente a BEAM). Demostramos también la trazabilidad del proceso de compilación al pasar de una semántica a otra.
- En el último capítulo se presenta un resumen del trabajo y su conclusión.
- En el Apéndice A desarrollamos las pruebas que planteamos en el capítulo 5 para probar la corrección del proceso de compilación de expresiones.
- En el Apéndice B detallamos la lista de funciones a implementar para el correcto funcionamiento de Shen.

Capítulo 2

Shen

2.1. Comienzos de Shen

Los inicios de Shen datan de 1989, cuando Mark Tarver comienza a trabajar en un lenguaje de la familia de Lisp completamente tipado, que permite especificar tipos en forma de reglas de deducción. Este lenguaje pasó a ser llamado SEQUEL (*SEQUEnt processing Language* [3]) y fue escrito enteramente en Lisp. SEQUEL fue el predecesor para gran parte de la funcionalidad de Shen, en particular, el uso de la notación de Gentzen para formular las reglas para especificar tipos.

Luego, al reconocer varias deficiencias de SEQUEL como lenguaje – por ser ineficiente, por no tener una semántica clara definida y por no soportar las características de un programa consistente con el Cálculo Lambda – Mark Tarver creó un nuevo lenguaje: Qi. Este lenguaje, luego de pasar por varias versiones, terminó siendo dejado de lado por unos años, para luego ser retomado por Tarver nuevamente.

Tanto SEQUEL como Qi fueron implementados en Common Lisp. La idea de Tarver era lograr que un lenguaje fuera implementado por el menor número de primitivas posibles, por lo que implementó una versión similar a Qi a partir de un nuevo conjunto de primitivas (independientes de Common Lisp). Este lenguaje de primitivas se compuso de 57 primitivas y terminó siendo llamado Kλ (pronunciado "KLambda").

La utilización de Kλ para implementar el lenguaje diseñado por Tarver dio lugar a un nuevo lenguaje, sucesor de Qi, creado en 2011: Shen.

2.2. Sintaxis y semántica de Shen

Shen se describe como un lenguaje de la familia de Lisp, tipado y fácilmente portable a otras tecnologías a través de Kλ. Las principales características de Shen son [4]:

- *Pattern-matching*
- Macros para definir "lenguajes de dominio específico"
- Evaluación perezosa (*lazy*) opcional
- Chequeo estático de tipos basado en el Cálculo de Secuentes ([5]), uno de los sistemas de mayor poder para especificar la asignación de tipos en lenguajes funcionales
- Portabilidad sobre varios lenguajes (a través de los puertos de la comunidad)
- Prolog integrado como parte del lenguaje
- Un "compiler-compiler" integrado como parte del lenguaje

A continuación describiremos brevemente y de forma informal cada una de estas características junto con ejemplos del uso que se le da a Shen para luego entender el funcionamiento de Kλ.

2.2.1. Tipos básicos

Los tipos básicos de Shen son símbolos, cadenas de caracteres (*strings*), números y *booleanos*. Los objetos de estos tipos evalúan a sus correspondientes valores de forma directa, ya que son valores irreducibles. Algunos ejemplos son:

- Símbolos – `sol`, `merge_sort`, `sum+`
- Strings – `''ham''`
- Números – `5`, `6.5`, `19278381e18`
- Booleanos – `true`, `false`

2.2.2. Aplicación de funciones

La sintaxis para llamar a funciones con cero o más parámetros es la siguiente: `(f x y)`, donde `f` es la función y `x` e `y` son los parámetros. La forma en que se evalúan las llamadas se denomina *applicative strict*, que significa que primero se evalúan los parámetros de derecha a izquierda para luego aplicar la función.

Adicionalmente, en Shen las funciones están "currificadas", lo que implica que evaluar `(f x y)` es equivalente a evaluar primero `(f x)` y luego llamar a la función resultante con `y` (`((f x) y)`).

2.2.3. Secuencias

Shen tiene (además de strings) tres tipos para describir secuencias:

- Listas – `[]`, `[1 2 3]`, `[1 | [2 3]]`. Estos dos últimos representan el mismo valor.
- Tuplas – Representados de la forma `(@p <valores de la tupla)`. Si bien a nivel de sintaxis se puede representar una tupla de más de dos elementos, Shen interpreta la tupla `(@p a b c)` como una tupla de la forma `(@p a (@p b c))`.
- Vectores – Representados de la forma `(@v a b .. n v)`, donde `a b .. n` son 0 o más elementos del vector y `v` es otro vector. El vector vacío se representa como `<>`. Esta estructura es una estructura mutable, se puede modificar el contenido de un vector a través de la operación `(vector->v n x)` donde `v` es un vector y `n` es el índice ($0 < n \leq size(v)$).

Algunas de las operaciones que se pueden realizar sobre estas secuencias se ejemplifican a continuación:

```
> (head [1 2 3])
1
> (tail [1 2 3])
[2 3]
> (fst (@p a b c))
```

```
a
> (snd (@p a b c))
(@p b c)
> (@v 1 2 <>)
<1 2>
> (@v 1 2 (@v 3 4 <>))
<1 2 3 4>
> (@v a b <>)
<a b>
> (<-vector (@v a b <>) 2)
b
```

2.2.4. Definiciones locales y globales

Las definiciones locales se introducen con la primitiva `let`, que asigna una o más variables y devuelve una expresión resultante.

```
> (let A 1
       B (+ A 1)
       (+ A B))
3
```

A su vez, Shen permite definiciones globales mutables, donde las funciones `set` y `get` asignan y obtienen los valores respectivamente.

```
> (set cantidad 1)
1
> (set cantidad (+ (get cantidad) 1))
2
> (get cantidad)
2
```

2.2.5. Funciones

Las funciones en Shen se definen a través de reglas de *pattern-matching* [1]. Para cada función se especifican n reglas que determinan su resultado.

```
> (define map
    _ [] -> []
    F [H | T] -> [(F H) | (map F T)])
map
> (map (+ 3) [1 2 3])
[4 5 6]
```

Las funciones en Shen pueden no ser totales y para cada regla se puede especificar una guarda opcional, utilizando `where`, cuya expresión devuelve un *booleano*.

```
> (define map
    F L where (cons? L) -> \\ cons? devuelve true si la lista no es vacía
      [(F (head L)) | (map F (tail L))]
    _ _ -> [])
map
```

Adicionalmente, Shen permite lo que el autor llama "reglas de reescritura", que permiten utilizar la misma variable del lado de la izquierda de la flecha como parte del patrón de *pattern-matching*. La aplicación de esa regla resultará exitosa si las múltiples ocurrencias de una misma variable son ligadas a el mismo valor.

```
> (define identical
    X X -> true
    _ _ -> false)
identical
```

Por último, Shen también permite definir funciones anónimas ("abstracciones lambda") de la siguiente forma:

```
> (/. X Y (* X (+ Y 1)))
#<FUNCTION (LAMBDA (X)) {100366004B}>
```

donde X e Y son los parámetros y `(* X (+ Y 1))` es el cuerpo de la función. Ejemplo:

```
> (map (/. X (* X -1)) [1 2 3])
[-1 -2 -3]
```

2.2.6. eval

Otra de las funciones que expone Shen para su uso es *eval*, que recibe una expresión E y evalúa la expresión resultante de transformar las listas que recibe por parámetros a expresiones delimitadas por paréntesis. Ejemplo:

```
> (eval [let X 3 [+ X 1]])
4
```

2.2.7. Chequeo de tipos estático basado en el Cálculo de Secuentes

Una de las principales características de Shen es su sistema de tipos. Un sistema de tipos se utiliza para clasificar las expresiones y asegurar que todas cumplen con ciertas reglas dadas.

En Shen existen varias formas de utilizar y crear nuevos tipos. En primer lugar existen los *tipos base*, que son: *symbol* (`banana`, `p12`), *string* (`‘‘dia miércoles’’`), *boolean* (`true`, `false`) y *number* (`1`, `2.12`).

Luego existen los operadores de tipos, que construyen nuevos tipos a partir de otros. Shen incluye los siguientes operadores:

- *list* – Donde dado un tipo A, `list A` es el tipo de las listas cuyos elementos tienen tipo A.
- *vector* – Donde dado un tipo A, `vector A` es el tipo de los vectores cuyos elementos tienen tipo A.
- $\rightarrow$ – Donde dado dos tipos A y B, $A \rightarrow B$ es el tipos de las funciones que reciben un elemento del tipo A y retornan un elemento del tipo B.
- *lazy* – Donde dado un tipo A, `lazy A` es el tipo de los objetos “lazy” que al ser evaluados retornan un objeto del tipo A.

A su vez, a través de Shen se pueden definir nuevos tipos utilizando reglas de refinamiento, que permiten derivar secuentes de la forma $\Delta >> e : \alpha$, donde Δ es una lista de expresiones que se denominan “antecedentes” y $e : \alpha$ es un predicado que establece que la expresión e tiene tipo α, denominado

"consecuente". Esto determina que dado los antecedentes, se cumple el predicado $e : \alpha$. Los tipos definidos de esta forma se llaman "tipos deductivos".

Para definir un nuevo tipo debemos establecer las reglas que permiten construir elementos de ese tipo. Como primer ejemplo, para definir un tipo enumerable en Shen, se realiza de la siguiente forma:

```
(datatype dia
  ___________
  lunes : dia;

  ___________
  martes : dia;

  ___________
  miercoles : dia;)
```

En ninguna de las reglas de este ejemplo existen antecedentes. La primer regla, por ejemplo, establece que `lunes` tiene el tipo `dia`.

Además de predicados de la forma $e : \alpha$, podemos definir también condiciones en las premisas de la regla utilizando `if` seguido de una expresión que retorna un *booleano*:

```
(datatype dia

  if (element? X [lunes martes miercoles])
  ________________________________________
  X : dia;)
```

Shen también permite definir tipos recursivos y polimórficos, como puede ser el tipo de las listas. Incluso podemos definir tipos con estructuras compuestas, utilizando la notación de listas. Por ejemplo, para definir un nuevo tipos de listas, utilizamos dos nuevos constructores: `lnil` que representa la lista vacía y `[lcons X XS]`, donde el primer elemento es el símbolo `lcons`, X es del tipo A y XS del tipo *lista* A.

```
(datatype lista

  ______________
```

```
  lnil : (lista A);

  X : A;
  XS : (lista A);
  -------------
  [lcons X XS] : (lista A);)
```

En este ejemplo ninguna de las hipótesis ni conclusiones utilizados tienen antecedentes en su expresión como secuente.

Uno puede utilizar valores de estos tipos libremente, pero en algunos casos será necesario indicar de qué tipo la expresión es:

```
> [lcons 1 lnil]
type error!

> [lcons 1 lnil] : (lista number)
[lcons 1 lnil] : (lista number)
```

Al definir funciones esto no será necesario debido a que todas ellas deben contener la especificación de su tipo. Sin embargo, en el siguiente ejemplo, utilizando el tipo que definimos para listas, no podemos inferir que dado `[lcons X XS] : (lista A)` entonces X es de tipo A.

```
> (define lmap
    {(A --> B) --> (lista A) --> (lista B)}
    F [lcons X XS] -> [lcons (F X) (lmap F XS)]
    F lnil -> lnil)
type error!
```

Para solucionar esto, debemos ajustar la definición de *lista* utilizando nuevos secuentes que indican que "si dado un valor `X : A` y un valor `XS : (lista A)` puedo probar P, entonces con un valor `[lcons X XS] : (lista A)` puedo probar P también":

```
(datatype lista

  -------------
  lnil : (lista A);
```

```
X : A;
XS : (lista A);
---------------
[lcons X XS] : (lista A);

X : A, XS : (lista A) >> P;
------------------------
[lcons X XS] : (lista A) >> P;)
```

Existen otras formas de definir tipos, incluyendo la posibilidad de definir tipos dependientes, subtipos y tipos negativos que no cubrimos en esta breve sinopsis de Shen. La intención fue mostrar el poder que tiene Shen en cuanto a su sistema de tipos.

2.2.8. Características adicionales

Existen también varias características de Shen que no se han mencionado. Éstas son más complejas pero de gran utilidad al momento de programar en Shen, pero no aportan para entender la compilación de Shen a Kλ. Estas características son:

- Excepciones – Permite lanzar excepciones con la función `error` y capturarlas con `trap-error` que contiene la función a evaluar si ocurre una excepción al evaluar la expresión. Por ejemplo, `(trap-error (error ‘‘Un mensaje de error’’) F)` evalúa a `(F ‘‘Un mensaje de error’’)`.
- Backtracking – Permite que las funciones “fallen” utilizando `(fail)` para caer en la siguiente guarda de la función que unifique.
- Macros – Permite definir macros de la forma (`defmacro` $< nombre >$ $P \rightarrow$ C), donde P y C son expresiones de Shen delimitadas con paréntesis balanceados representadas como listas, reemplazando el código con la estructura de P por código con la estructura de C.
- Compiler-compiler – Permite definir un compiler-compiler al estilo YACC [6] utilizando la notación BNF.
- Prolog – Provee una forma de definir reglas y hechos (*facts*) al estilo Prolog para luego realizar operaciones utilizando el código Shen regular.

- Paquetes – Permite definir paquetes para aislar la funcionalidad provista por una librería y prevenir el solapamiento de nombres de funciones exportadas.
- Funciones adicionales (streams, archivos) – Provee una librería standard para operar con I/O (archivos) a través de "streams" de entrada y salida.

A continuación, describiremos el lenguaje al que Shen compila: Kλ. Este compilador fue construido por el autor de Shen con la intención de lograr un lenguaje altamente portable. Como mencionamos anteriormente, para que Kλ sea fácilmente portable debe ser un lenguaje reducido: debe tener una sintaxis simple y estar definido por una semántica simple.

Capítulo 3

Kλ

Kλ, al igual que Shen, es un lenguaje de la familia de Lisp y como tal carece de la complejidad que gran cantidad de lenguajes populares hoy poseen. Kλ no es estáticamente tipado, por lo que no posee un sistema de tipos, simplemente está definido por su semántica y sintaxis.

3.1. Sintaxis

La sintaxis de Kλ es similar a lo de todos los lenguajes de la familia Lisp cuyos programas son formados por una secuencia de *S-expressions*. Una *S-expression* es una notación para representar estructuras de árbol, popularizada por Lisp y luego utilizada por nuevos lenguajes de programación (incluyendo Shen y Kλ).

La sintaxis de Kλ entonces está únicamente compuesta por las siguientes estructuras:

- Literales, que incluye números, símbolos, cadenas de texto (delimitados por comillas dobles) o la lista vacía ().
- Aplicaciones, que son *S-expressions* con n expresiones (n $\geq$ 0).
- Definiciones de la forma (`defun` *symbol* (*params*) *expression*), donde *params* es una lista (sin duplicados) de símbolos.

3.2. Sintaxis Abstracta

Para poder trabajar sobre la semántica operacional y los posibles programas de Kλ, definimos una sintaxis abstracta.

En particular, un programa en Kλ está compuesto por una secuencia de cero o más definiciones ($<d>$), que pueden ser definiciones de funciones (*defun*) o expresiones de Kλ que las llamaremos "expresiones de primer nivel".

Para definir la sintaxis abstracta partimos de los programas y declaraciones para luego definir las expresiones:

Programas $< p > ::=$ $< d_1 > \ \ldots \ < d_n >$ (Secuencia de declaraciones)

Declaraciones $< d > ::=$ (`defun` x $(< s_1 > \ \ldots \ < s_n >)$ $< e >$)
(Definición)
$|$ $< e >$
(Expresión de primer nivel)

Expresiones $< e > ::=$			
	$< n >$		(Número)
$\|$	$< s >$		(Símbolo)
$\|$	$< str >$		(String)
$\|$	$(< e_1 > \ \ldots \ < e_n >)$		(S-expression)

A partir de estas definiciones describiremos cómo formar programas válidos.

3.3. Semántica operacional

Como mencionamos anteriomente la semántica operacional de un lenguaje determina qué hace un programa del lenguaje al ejecutarse. Utilizando reglas de inferencia definiremos la semántica formal de Kλ, que será luego utilizada para construir el puerto del trabajo.

Actualmente, la semántica operacional de Kλ está determinada por el comportamiento de lo que en la documentación de Kλ llaman "57 primitivas" [7], que incluyen primitivas propias del lenguaje y funciones que provee

Kλ. Estas primitivas determinan las expresiones válidas de Kλ y definen la ejecución del lenguaje en su totalidad.

A partir de la sintaxis abstracta de Kλ, utilizaremos cuatro relaciones para definir su semántica operacional:

- Una relación que determina la *ejecución* de un programa ($\Downarrow_p$).
- Una relación que determina la *carga de funciones* de un programa ($\blacktriangleright$).
- Una relación que determina la *ejecución* de un programa bajo una tabla de funciones globales ($\underline{\Downarrow_p}$).
- Una relación que determina la *evaluación* de una expresión. ($\downarrow$).

Utilizaremos reglas de inferencia para definir estas relaciones. Éstas reglas son de la forma

$$\textit{nombre regla}\ \frac{P_1\ P_2\ \ldots\ P_n}{C,}\ \textit{condiciones}$$

donde las afirmaciones P_i son las premisas, C es la conclusión y las condiciones son predicados que se deben cumplir para que la regla pueda darse. [1]

Para definir cómo ejecutan o evalúan cada uno de los elementos de un programa utilizando este sistema de reglas de inferencia primero deberemos definir qué es un valor y qué es un contexto.

3.3.1. Valores

Al evaluar una expresión uno obtiene un valor. Este valor es irreducible (canónico), por lo que no se puede realizar más computos sobre el valor resultante y su representación es única.

En Kλ, los valores posibles son:

[1] Por una cuestión de legibilidad las premisas de todas las reglas utilizadas en este trabajo podrán ocupar varias líneas.

Valores $< v >::=$	$< n >$	(Número)
	$\mid < s >$	(Símbolo)
	$\mid < str >$	(String)
	$\mid$ `()`	(Lista vacía)
	$\mid \{(\texttt{lambda}\ s_{param}\ e_{body}), \gamma\}$	(Abstracción)
	$\mid \{(\texttt{freeze}\ e_{body}),\ \gamma\}$	(Valor "congelado")

Algunas aclaraciones:

- Las abstracciones tienen un solo parámetro.
- Los valores "congelados" se representarán como abstracciones sin parámetros. Esta es la única forma de construir abstracciones sin parámetros.
- Tanto las abstracciones como los valores "congelados" capturan el contexto actual de variables para luego ser utilizado cuando se aplica la abstracción. Esto es lo que se llama "clausura" y lo veremos luego de definir qué es un contexto.

Nótese que ninguno de estos términos puede ser reducido, es decir, no se pueden realizar más cómputos sobre ellos y son por lo tanto, lo que llamaremos los "valores" de Kλ.

3.3.2. Contexto

El contexto que definimos para trabajar sobre los programas en Kλ contiene una tabla que asocia nombres de variables (símbolos) a valores.

Las formas de construir un contexto (sus constructores) son los siguientes:

- Contexto vacío ($\epsilon : Ctx$) que retorna un contexto sin ninguna asociación.
- Asignación de variable ($\longleftarrow : Ctx \times (< s >, < v >) \rightarrow Ctx$), que asigna una variable s (símbolo) a un valor v y retorna el nuevo contexto con la asignación.

Este contexto representa una estructura de datos sobre la que definimos una única relación: @, que dado un contexto y un símbolo determina el valor del mismo si este se encuentra en el contexto (*Some* v) o *None* si no.

Para eso introducimos una nueva estructura *Option* <v> como:

$$Option < v > ::= None \mid Some < v >$$

Definimos ahora relación @, que determina si el valor asociado a una variable existe, utilizando la siguiente notación:

$$\gamma \ @ \ s \ \longrightarrow \ ov$$

que se lee como "el contexto γ está asociado a un valor opcional ov".

$$\text{lookup ctx empty} \ \frac{}{\epsilon \ @ \ s \ \longrightarrow None}$$

$$\text{lookup ctx head} \ \frac{}{(\gamma \ \longleftarrow \ (s, \ v)) \ @ \ s \ \longrightarrow Some \ v}$$

$$\text{lookup ctx tail} \ \frac{\gamma \ @ \ s \ \longrightarrow \ ov}{(\gamma \ \longleftarrow \ (s', \ v')) \ @ \ s \ \longrightarrow ov} \ s \ \neq \ s'$$

3.3.3. Clausura

Al definir nuevas funciones se utiliza la llamada "clausura léxica" (*lexically scoped closure*), donde se captura el contexto actual al momomento de definir la función para luego ser utilizado al evaluar su aplicación.

A nivel de la semántica de Kλ, esto implica que al definir una función se deberá asociar, junto con ella, un contexto que contenga estas variables

accesibles al momento de definir la función y sus valores. Para lograrlo, representamos al valor del tipo función como un par que contiene la definición de la función y el contexto dado por su clausura.

Definir este par *función-contexto* como valor del lenguaje surge porque las variables en Kλ son *statically scoped*, lo que significa que, al momento de evaluar la llamada a la función, el valor asociado a la variable es el valor que esa variable tenía al momento de definir la función [1].

Otra técnica utilizada para definir la semántica de un lenguaje cuando éste contiene variables es utilizando el principio de sustitución [8, 9], utilizado originalmente en el Cálculo de Lambda. Esta técnica lo que hace es, al momento de evaluar la asignación de una variable a un valor, sustituye todas las referencias a esta variable (que estén dentro de su *scope*) por el propio valor como expresión; de esta forma no se necesita llevar un contexto que contenga las asociaciones *variable-valor*. Sin embargo, esta forma de definir la semántica del lenguaje no aplica para aquellos lenguajes que tienen "efectos secundarios" (*side-effects*) al evaluar expresiones. Éste es el caso de Kλ, donde sustituir todas las ocurrencias de una variable por una llamada a una función (por ejemplo, abrir un archivo) tendría un comportamiento distinto que evaluando la expresión primero y guardando su valor en el contexto para accederlo cuando se evalúe la variable.

Es por este motivo entonces que utilizamos un contexto para definir la evaluación de expresiones y asociamos junto con los valores de las funciones un contexto accesible al momento de aplicar la función.

3.3.4. Tabla de funciones globales

En Kλ las funciones definidas a través de las definiciones de función en el primer nivel de un programa tienen acceso global. Cualquier parte del programa puede acceder a ellas; incluso el cuerpo de una función puede acceder a esa misma función.

Para poder representar las asociaciones entre los nombres de las funciones (símbolos) y las funciones mismas deberemos definir una nueva estructura, en este caso, una tabla de asociaciones.

De la misma forma que con el contexto, esta tabla de funciones globales representa una estructura de datos (con constructores) sobre la que se pueden definir relaciones, con la diferencia que el contexto asocia símbolos con valores y la tabla de funciones asocia símbolos con expresiones. Utilizaremos la una

nueva notación para la relación *@!*. Esta relación está definida por un contexto (γ) un símbolo (s) y una expresión no opcional (e_{fun}).

La definición de esta relación es la siguiente:

$$\text{map lookup! head}\ \frac{}{(\gamma \longleftarrow (s,\ e_{fun}))\ @!\ s \longrightarrow e_{fun}}$$

$$\text{map lookup! tail}\ \frac{\gamma\ @!\ s \longrightarrow e}{(\gamma \longleftarrow (s',\ e_{fun}))\ @!\ s \longrightarrow e}\ s \neq s'$$

Nótese que, a diferencia de la relación *@* del contexto, la expresión e_{fun} asociada al símbolo s debe estar definida para formar parte de la relación *@!*. *@!* por lo tanto nunca estará definida para símbolos que no se encuentran en la tabla.

3.3.5. Ejecución de programas

En primer lugar partimos de un programa, compuesto por una secuencia de cero o más declaraciones. Decimos que un programa ejecuta cuando "dado la tabla que contiene todas las definiciones de función del programa, todas sus expresiones de primer nivel evalúan a un valor".

Para ello, primero se debe construir la tabla conteniendo las definiciones de funciones. Llamaremos a este proceso de construcción "carga de funciones globales".

Carga de funciones globales

La carga de funciones globales involucra agregar todas las funciones del programa a la tabla de funciones globales. Esto se logra a través de una nueva relación ($\blacktriangleright$) que construye la tabla a partir de programas y sus declaraciones.

Esta relación está dada por tres elementos:

- Una tabla f.
- Un programa p, compuesto por una secuencia de definiciones.
- Una nueva tabla f' resultante de cargar las nuevas definiciones de función del programa p.

Para el caso que se quiera cargar una función en la tabla, debemos introducir una nueva función: *currify*, que, dada una lista de nombres de parámetros y el cuerpo de la función, retorna expresiones lambda anidadas que culminan retornado la el cuerpo de la función como expresión. Por ejemplo:

```
(currify (A B) (+ A B)) = (lambda A (lambda B (+ A B)))
```

La relación es entonces es la siguiente:

$$\text{load defun}\ \frac{f \ \longleftarrow\ (x,\ (currify\ (s_1\ \ldots\ s_n)\ e_{body}))\ \vdash d_2\ \ldots\ d_n\ \blacktriangleright\ f'}{f \vdash (\texttt{defun}\ x\ (s_1\ \ldots\ s_n)\ e_{body})\ d_2\ \ldots\ d_n\ \blacktriangleright\ f'}$$

$$\text{ignore top-level-expression}\ \frac{f\ \vdash d_2\ \ldots\ d_n\ \blacktriangleright\ f'}{f\ \vdash e\ d_2\ \ldots\ d_n\ \blacktriangleright\ f'}$$

$$\text{load nil}\ \frac{}{f\ \vdash \blacktriangleright\ f}$$

Ejecución de expresiones de primer nivel de un programa

Dada una tabla de funciones globales, uno puede ejecutar las expresiones de primer nivel de un programa. Para esto definimos una nueva relación ($\Downarrow_p$) determinada por dos elementos:

- Una tabla de funciones globales *f*.
- Un programa p, compuesto por una secuencia de definiciones.

La notación que utilizaremos para esta relación es la siguiente:

$$f \vdash p \Downarrow_p$$

que se lee como "el programa p ejecuta bajo la tabla de funciones globales f".

$$\text{exec defun}\ \frac{f \vdash d_2\ \ldots\ d_n \Downarrow_p}{f \vdash (\texttt{defun}\ x\ (p_1\ \ldots\ p_n)\ e_{body})\ d_2\ \ldots\ d_n \Downarrow_p}$$

$$\text{exec top-level-expression}\ \frac{\langle \epsilon,\ f \rangle \vdash e\ \downarrow\ S\langle v \rangle \qquad d_2\ \ldots\ d_n \Downarrow_p}{f \vdash e\ d_2\ \ldots\ d_n \Downarrow_p}$$

$$\text{exec nil}\ \frac{}{f \vdash \Downarrow_p\ f}$$

Para el caso de la regla *exec top-level-expression* se utiliza la relación $\downarrow$ que define la evaluación de expresiones que veremos más adelante. En este caso, la relación se lee como "la expresión e evalúa bajo el contexto vacío y la tabla de funciones f a un valor exitoso v". Nótese que este valor v se ignora en la conclusión, dado que no se utiliza para ejecutar el programa.

Ejecución de un programa

Ahora sí estamos en condiciones de determinar la relación para ejecutar un programa. Esta relación tiene una única regla que surge de la definición de ejecución de un programa: "dado la tabla que contiene todas las definiciones de funciones del programa, todas sus expresiones de primer nivel evalúan a un valor exitoso". La notación que utilizaremos es la siguiente:

$$p\ \Downarrow_p$$

que se lee como "un programa ejecuta".

La única forma de definir un elemento de esta relación es a través de la siguiente regla:

$$\text{exec prog}\ \frac{\epsilon \vdash p\ \blacktriangleright\ f \qquad f \vdash p \Downarrow_p}{p\ \Downarrow_p}$$

3.3.6. Evaluación de expresiones

A partir de haber definido lo que representa un valor en Kλ, estamos en condiciones de definir la evaluación de expresiones. La relación que utilizaremos ($\downarrow$) para definir la evaluación de una expresión es una relación de cuatro elementos:

- Un contexto γ, que contiene las asociaciones de nombres de variables a sus correspondientes valores.
- Una tabla de funciones globales *f*.
- Una expresión *e*.
- Un valor resultado r que puede ser un valor exitoso o un error.

Una expresión puede evaluar a una forma exitosa que retorna un valor o a un error que se propaga por el flujo de llamadas del programa hasta ser capturado por una expresión `trap-error`.

La notación que utilizaremos para esta relación en el caso de éxito es la siguiente (la S surge de "*Success*" o "éxito"):

$$\langle \gamma,\ f \rangle \vdash e \downarrow S\langle v \rangle$$

que se lee como "la expresión e evalúa exitosamente bajo el contexto γ y la tabla f a un valor v".

La notación que utilizaremos para esta relación en el caso de error es la siguiente:

$$\langle \gamma,\ f \rangle \vdash e \downarrow E\langle str \rangle$$

que se lee como "la expresión `e` evalúa bajo el contexto γ a un error `s`.

Las expresiones en Kλ se clasifican en tres categorías, cuyo comportamiento para evaluar difiere:

- Expresiones atómicas (números, símbolos, *strings* y la lista vacía). El valor de la evaluación de estas expresiones es la propia expresión representada como valor.

- Expresiones de "forma especial" (así es como la llama el autor de Kλ), que se representan como "*S-expressions*" y cuya evaluación depende de su estructura, (por ejemplo, expresiones `lambda`, `if`, `cond`).
- Aplicaciones de funciones que se representan como "*S-expressions*", donde el primer argumento es la función que debe ser aplicada. Para estos casos la forma en que se evalúan las llamadas a funciones (al igual que Shen) es "*applicative strict*", lo que significa que primero se evalúan los parámetros de derecha a izquierda para luego aplicar la función.

A continuación se define la evaluación para estos tres tipos de expresiones.

3.3.6.1. Evaluación de expresiones atómicas

La evaluación de expresiones atómicas retorna el valor asociado a esa expresión.

$$\text{kl num}\ \frac{n}{\langle \gamma,\ f \rangle \vdash n\ \downarrow\ S\langle n \rangle}\ \text{n número}$$

$$\text{kl string}\ \frac{s}{\langle \gamma,\ f \rangle \vdash s\ \downarrow\ S\langle s \rangle}\ \text{s string}$$

$$\text{kl nil}\ \frac{\texttt{()}}{\langle \gamma,\ f \rangle \vdash \texttt{()}\ \downarrow\ S\langle \texttt{()} \rangle}$$

Para el caso de los símbolos, se debe tener en cuenta el contexto. Si dentro del contexto existe la asociación del nombre de variable a a un valor v, entonces la expresión a evalúa a $S<v>$ y no al símbolo. En cualquier otro caso, la expresión evalúa al símbolo.

$$\text{kl sim}\ \frac{\gamma\ @\ s\ \longrightarrow None}{\langle \gamma,\ f \rangle \vdash s\ \downarrow\ S\langle s \rangle}\ \text{s símbolo}$$

$$\text{kl sim-var}\ \frac{\gamma\ @\ s\ \longrightarrow Some\ v}{\langle \gamma,\ f \rangle \vdash s\ \downarrow\ S\langle v \rangle}\ \text{s símbolo}$$

3.3.6.2. Evaluación de expresiones de "forma especial"

Las "formas especiales" en Kλ son aquellas "*S-expressions*" que comienzan con una palabra reservada del lenguaje y tienen un formato determinado para sus sub-expresiones. Estas formas especiales son: `if`, `cond`, `and`, `or`, `lambda`, `let`, `freeze` y `trap-error`.

Evaluación de expresiones `if`

La "forma especial" del `if` es "corto-circuitada", es decir, no evalúa la expresión del consecuente ni la alternativa hasta no evaluar primero la condición para determinar cuál de las otras dos expresiones evaluar.

$$\text{kl if-true}\ \frac{\langle \gamma,\ f\rangle \vdash e\ \downarrow S\langle \texttt{true}\rangle \qquad \langle \gamma,\ f\rangle \vdash e_{true} \downarrow r}{\langle \gamma,\ f\rangle \vdash (\texttt{if}\ e\ e_{true}\ e_{false})\ \downarrow\ r}$$

$$\text{kl if-false}\ \frac{\langle \gamma,\ f\rangle \vdash e\ \downarrow S\langle \texttt{false}\rangle \qquad \langle \gamma,\ f\rangle \vdash e_{false} \downarrow r}{\langle \gamma,\ f\rangle \vdash (\texttt{if}\ e\ e_{true}\ e_{false})\ \downarrow\ r}$$

$$\text{kl if-error}\ \frac{\langle \gamma,\ f\rangle \vdash e\ \downarrow E\langle str\rangle}{\langle \gamma,\ f\rangle \vdash (\texttt{if}\ e\ e_{true}\ e_{false})\ \downarrow\ E\langle str\rangle}$$

Evaluación de expresiones `cond`

La "forma especial" de `cond` está compuesta una serie de pares, donde el primer elemento es una condición y el segundo es la expresión que se debe evaluar si la condición evalúa a `true`. El primer par cuya condición evalúe a `true` es el que retorna la evaluación de su expresión.

$$\text{kl cond-true}\ \frac{\langle \gamma,\ f\rangle \vdash econd_1\ \downarrow S\langle \texttt{true}\rangle \qquad \langle \gamma,\ f\rangle \vdash ebody_1 \downarrow r}{\langle \gamma,\ f\rangle \vdash (\texttt{cond}\ (econd_1\ ebody_1)\ \overline{rest})\ \downarrow\ r}$$

$$\text{kl cond-false}\ \frac{\langle \gamma,\ f\rangle \vdash econd_1\ \downarrow S\langle \texttt{false}\rangle \qquad \langle \gamma,\ f\rangle \vdash (\texttt{cond}\ \overline{rest}) \downarrow r}{\langle \gamma,\ f\rangle \vdash (\texttt{cond}\ (econd_1\ ebody_1)\ \overline{rest})\ \downarrow\ r}$$

$$\text{kl cond-error}\ \frac{\langle \gamma,\ f\rangle \vdash econd_1\ \downarrow E\langle str\rangle}{\langle \gamma,\ f\rangle \vdash (\texttt{cond}\ (econd_1\ ebody_1)\ \overline{rest})\ \downarrow\ E\langle str\rangle}$$

El caso en el que `cond` está vacío (no tiene más ramas para evaluar) no se define ya que se considera un programa mal definido. Este caso no se considera un error de evaluación recuperable.

Evaluación de operadores `and` y `or`

Estos operadores, al igual que la "forma especial" `if` son "corto-circuitados" – no evalúan su segundo parámetro si ya se conoce el resultado evaluando el primero.

$$\text{kl and-true}\ \frac{\langle \gamma,\ f\rangle \vdash e_1\ \downarrow S\langle\texttt{true}\rangle \qquad \langle \gamma,\ f\rangle \vdash e_2 \downarrow r}{\langle \gamma,\ f\rangle \vdash (\texttt{and}\ e_1\ e_2)\ \downarrow\ r}$$

$$\text{kl and-false}\ \frac{\langle \gamma,\ f\rangle \vdash e_1\ \downarrow S\langle\texttt{false}\rangle}{\langle \gamma,\ f\rangle \vdash (\texttt{and}\ e_1\ e_2)\ \downarrow\ S\langle\texttt{false}\rangle}$$

$$\text{kl and-error}\ \frac{\langle \gamma,\ f\rangle \vdash e_1\ \downarrow E\langle str\rangle}{\langle \gamma,\ f\rangle \vdash (\texttt{and}\ e_1\ e_2)\ \downarrow\ E\langle str\rangle}$$

$$\text{kl or-true}\ \frac{\langle \gamma,\ f\rangle \vdash e_1\ \downarrow S\langle\texttt{true}\rangle}{\langle \gamma,\ f\rangle \vdash (\texttt{or}\ e_1\ e_2)\ \downarrow\ S\langle\texttt{true}\rangle}$$

$$\text{kl or-false}\ \frac{\langle \gamma,\ f\rangle \vdash e_1\ \downarrow S\langle\texttt{false}\rangle \qquad \langle \gamma,\ f\rangle \vdash e_2 \downarrow r}{\langle \gamma,\ f\rangle \vdash (\texttt{or}\ e_1\ e_2)\ \downarrow\ r}$$

$$\text{kl or-error}\ \frac{\langle \gamma,\ f\rangle \vdash e_1\ \downarrow E\langle str\rangle}{\langle \gamma,\ f\rangle \vdash (\texttt{or}\ e_1\ e_2)\ \downarrow\ E\langle str\rangle}$$

Evaluación de expresiones `lambda`

En Kλ, uno puede definir una expresión `lambda`, que obligatoriamente debe recibir un único parámetro:

$$\text{kl lambda}\ \frac{}{\langle \gamma,\ f\rangle \vdash (\texttt{lambda}\ s_{param}\ e_{body})\ \downarrow\ S\langle\{(\texttt{lambda}\ s_{param}\ e_{body}),\ \gamma\}\rangle}$$

Evaluación de expresiones `let`

La extructura de `let` se utiliza para ligar una variable a un valor, cuya asignación sólo tiene validez dentro del cuerpo del `let`.

$$\text{kl let}\ \frac{\langle \gamma,\ f\rangle \vdash e\ \downarrow S\langle v\rangle \qquad \gamma \longleftarrow (s_{var},\ v)\ \equiv\ \gamma' \qquad \langle \gamma',\ f\rangle \vdash e_{body}\ \downarrow r}{\langle \gamma,\ f\rangle \vdash (\texttt{let}\ s_{var}\ e\ e_{body})\ \downarrow\ r}$$

$$\text{kl let-error}\ \frac{\langle \gamma,\ f\rangle \vdash e\ \downarrow E\langle str\rangle}{\langle \gamma,\ f\rangle \vdash (\texttt{let}\ s_{var}\ e\ e_{body})\ \downarrow\ E\langle str\rangle}$$

Evaluación de "valores congelados"

Los "valores congelados" son expresiones que se dejan sin evaluar, para luego ser evaluadas cuando se desee. En Kλ, estos valores se representan como funciones sin ningún parámetro. A modo de ejemplo, la siguiente expresión evalúa al número `3`:

```
(let (V1 (freeze (+ 1 2)))
     (V1))
```

La semántica de `freeze` es entonces:

$$\text{kl freeze}\ \frac{}{\langle \gamma,\ f\rangle \vdash (\texttt{freeze}\ e_{body})\ \downarrow\ S\langle\{(\texttt{freeze}\ e_{body}),\ \gamma\}\rangle}$$

Evaluación de expresiones `trap-error`

La estructura `trap-error` está definida por una expresión y un manejador (*handler*), que se aplicará como función en caso que ocurra una excepción al evaluar la expresión. Estas excepciones son lanzadas por la función primitiva `simple-error`.

$$\text{kl trap-error-success}\ \frac{\langle \gamma,\ f\rangle \vdash e\ \downarrow S\langle v\rangle}{\langle \gamma,\ f\rangle \vdash (\texttt{trap-error}\ e\ e_{handler})\ \downarrow\ S\langle v\rangle}$$

$$\text{kl trap-error-failure}\ \frac{\langle \gamma,\ f\rangle \vdash e\ \downarrow E\langle str\rangle \qquad \langle \gamma,\ f\rangle \vdash (e_{handler}\ str)\ \downarrow r}{\langle \gamma,\ f\rangle \vdash (\texttt{trap-error}\ e\ e_{handler})\ \downarrow\ r}$$

Para este último caso, al aplicar $e_{handler}$ como expresión representamos la cadena de caracteres como una expresión (no como un valor).

3.3.6.3. Evaluación de aplicación de funciones

Cuando una "*S-expression*" no tiene ninguna de las estructuras anteriores, se considera a esta expresión como una aplicación de función. La estructura de una "*S-expression*" cuando es vista como una aplicación debe contener al menos un elemento (si no se considera una lista vacía). El primer elemento de esta secuencia de expresiones dentro de la "*S-expression*" es llamado "operador", mientras que el resto de los elementos son llamados "operandos".

Cuando una aplicación tiene más de un operando se toma como varias aplicaciones anidadas, donde se asocian las expresiones a la izquierda. Por ejemplo: $(e_1\ e_2\ e_3\ e_4)$ se evaluará como $(((e_1\ e_2)\ e_3)\ e_4)$. Esto lo definimos con la siguiente regla:

$$\text{kl app-left-assoc}\ \frac{\langle \gamma,\ f\rangle \vdash ((e_1\ e_2)\ e_3\ \ldots\ e_n)\ \downarrow\ r}{\langle \gamma,\ f\rangle \vdash (e_1\ e_2\ e_3\ \ldots\ e_n)\ \downarrow\ r}\ n\ >\ 2$$

Dependiendo del operador, la forma de evaluación será distinta: si es un símbolo se considera una aplicación de función global, si es cualquier otra expresión se considera una aplicación de función lambda.

Evaluación de aplicación de funciones globales

Las aplicaciones de funciones globales (definidas en el *top-level* del programa con `defun`) están definidas por una "*S-expression*", donde el primer elemento es un símbolo y a su vez no es una variable dentro del contexto. [2]

$$\text{kl eval-global-fun}\ \frac{\gamma\ @\ s\ \longrightarrow\ None \qquad f\ @!\ s\ \longrightarrow e_{fun} \qquad \langle \gamma,\ f\rangle \vdash (e_{fun}\ e)\ \downarrow\ r}{\langle \gamma,\ f\rangle \vdash (s\ e)\ \downarrow\ r}$$

[2]Las expresiones cuyo primero elemento evalúa a un símbolo como valor no son consideradas llamadas a funciones globales.

En este caso, se reemplaza el símbolo de la función global por el cuerpo de la función *currificado* que se introdujo en la tabla de funciones globales.

Evaluación de aplicación de funciones lambda

Este caso se da cuando el operador evalúa a una función lambda y el número de operadores es mayor a cero.

Como las expresiones lamdba sólo reciben un único parámetro, aplicar un valor lambda con más de un operando implica aplicar la función con el primer operando y luego aplicar el resultado al resto de los operandos. Si se tiene más de un argumento la aplicación de una función lambda deberá retornar una nueva función.

$$\text{kl app-lambda}\ \frac{\langle \gamma,\ f\rangle \vdash e_1\ \downarrow\ S\langle\{(\texttt{lambda}\ s_{param}\ e_{body}),\ \gamma'\}\rangle \qquad \gamma' \longleftarrow (s_{param},\ v_2)\ \equiv\ \gamma'' \qquad \langle \gamma,\ f\rangle \vdash e_2\ \downarrow\ S\langle v_2\rangle \qquad \langle \gamma'',\ f\rangle \vdash e_{body}\ \downarrow\ r}{\langle \gamma,\ f\rangle \vdash (e_1\ e_2)\ \downarrow\ r}$$

A su vez, si el operador evalúa a un error, el mismo es el resultante de evaluar toda la expresión. De la misma forma, si el operando no evalúa a un error pero sí su parámetro, éste error es el resultante de evaluar toda la expresión.

$$\text{kl app-lambda-e1-error}\ \frac{\langle \gamma,\ f\rangle \vdash e_1\ \downarrow\ E\langle str\rangle}{\langle \gamma,\ f\rangle \vdash (e_1\ e_2)\ \downarrow\ E\langle str\rangle}$$

$$\text{kl app-lambda-e2-error}\ \frac{\langle \gamma,\ f\rangle \vdash e_1\ \downarrow\ S\langle\{(\texttt{lambda}\ s_{param}\ e_{body}),\ \gamma'\}\rangle \qquad \langle \gamma,\ f\rangle \vdash e_2\ \downarrow\ E\langle str\rangle}{\langle \gamma,\ f\rangle \vdash (e_1\ e_2)\ \downarrow\ E\langle str\rangle}$$

A modo de ejemplo, la siguiente expresión en Kλ evalúa al número `3`:

```
(let (F (lambda X (lambda Y (+ X Y))))
     (F 1 2))
```

Evaluación de aplicación de "valores congelados"

Este caso se presenta cuando existe una aplicación de función sin parámetros, es decir, cuando la "*S-expression*" sólo contiene la expresión del operador.

Evaluar una expresión con esta forma es evaluar primero la expresión del operador y luego (en caso de que no haya error) evaluar el valor congelado resultante.

$$\text{kl app-freeze}\ \frac{\langle \gamma,\ f\rangle \vdash e\ \downarrow\ S\langle\{(\texttt{freeze}\ e_{body}),\ \gamma'\}\rangle \qquad \langle \gamma',\ f\rangle \vdash e_{body}\ \downarrow\ r}{\langle \gamma,\ f\rangle \vdash (e)\ \downarrow\ r}$$

$$\text{kl app-freeze-error}\ \frac{\langle \gamma,\ f\rangle \vdash e\ \downarrow\ E\langle str\rangle}{\langle \gamma,\ f\rangle \vdash (e)\ \downarrow\ E\langle str\rangle}$$

3.4. Funciones y símbolos globales

Además de implementar las 9 "formas especiales" de expresiones y la aplicación de funciones, para lograr el correcto funcionamiento de Kλ se debe implementar adicionalmente 48 funciones y símbolos globales predefinidos como parte del puerto a construir.

Los símbolos globales en Kλ se almacenan en una tabla mutable que los asocia con valores de Kλ. Para modificar y obtener estos valores se debe acceder a través de dos funciones (primitivas) que deberán ser implementadas: `set` y `value`. Esta tabla se comporta igual que las definiciones globales en Shen (2.2.4), y se utiliza al compilar las funciones `set` y `get` de Shen.

La lista de las funciones y símbolos globales y su comportamiento se detalla en el Apéndice B.

3.5. Shen a Kλ

Si bien definimos la semántica de Shen de manera informal, ésta está dada por su compilación a Kλ, es decir, a través de la compilación a un "lenguaje *kernel*".

En la figura 3.1 se muestra un diagrama de los elementos que forman un puerto de Shen:

1. El lenguaje de origen es Shen.
2. El compilador de Shen a Kλ que está dado por la propia librería estándar de Shen, a través de la función de la librería estándar de Shen *shen->kl*.
3. El lenguaje intermedio es Kλ.
4. El compilador o intérprete es aquel que deberá ser desarrollado por el puerto.
5. El ambiente de ejecución o plataforma sobre la que corre el código compilado o intérprete del puerto.

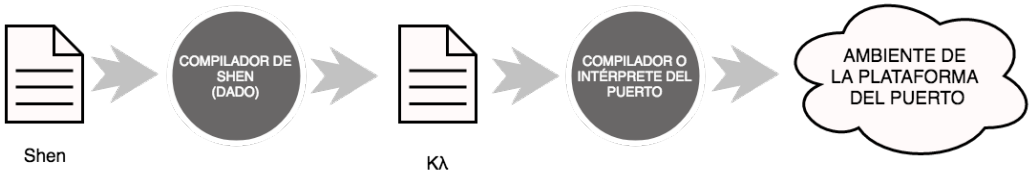

Figura 3.1: Estructura de un puerto de Shen

Esta librería en Shen (una vez compilada por algún puerto a Kλ) puede ser tomada por el puerto para ser usada para compilar archivos de Kλ a Shen. Para este trabajo se utilizó el puerto de Common Lisp [10] para compilar los archivos de la librería estándar de Shen a Kλ.

A modo de ejemplo, proveemos la compilación de Shen a Kλ de la función `map` y `identical` que definimos al definir Shen:

```
(define map
  _ [] -> []
  F [H | T] -> [(F H) | (map F T)])

(define identical
  X X -> true
  _ _ -> false)
```

compila a

```
(defun map (F L)
  (cond ((= [] L) [])
        ((cons? L) (cons (F (hd L)) (map F (tl L))))))

(defun identical (V1 V2)
  (cond ((= V1 V2) true)
        (true false)))
```

Capítulo 4

Erlang

Erlang es un lenguaje que surge en 1986 con el fin de facilitar el desarrollo de aplicaciones para telefonía. Actualmente se utiliza en ambientes de producción dentro de sistemas distribuidos de gran escala, debido a su robustez, capacidad de recuperación y facilidad para construir programas concurrentes sin tener en cuenta los distintos casos de borde que introduce la concurrencia [11, 12].

El enfoque principal de este trabajo será utilizar Erlang como lenguaje al que portar Kλ y, adicionalmente, como herramienta para construir el puerto (el compilador es un programa en Erlang).

Dejando de lado la concurrencia, Erlang resulta ser un lenguaje de sintaxis y semántica simple, que permite ser fácilmente especificado utilizando reglas de inferencia, similar a como lo hicimos con Kλ.

La especificación de la sintaxis y semántica operacional en Erlang de este trabajo cubrirá exclusivamente aquellas primitivas que fueron utilizadas para realizar el puerto.

4.1. La Máquina Virtual de BEAM

Los módulos Erlang son compilados a BEAM, y serán luego interpretados por la máquina virtual de BEAM para ser ejecutados. Por este motivo decimos que Erlang es también un lenguaje portable a través de la técnica de utilización de puertos. En este caso, el lenguaje intermedio es BEAM y el intérprete que lo ejecuta es la Máquina Virtual de BEAM.

En la figura 4.1 se muestra un diagrama de los elementos que forman el ecosistema de Erlang y el flujo que realizan los programas para ser ejecutados. Estos elementos son:

1. El código del lenguaje de origen, en este caso Erlang.
2. El compilador de Erlang a BEAM que forma parte de la librería estándar de Erlang y no depende de la plataforma.
3. El código de lenguaje intermedio, en este caso BEAM.
4. La máquina virtual de BEAM que interpreta el código BEAM en la plataforma. Como este intérprete depende de la plataforma, deberá ser construido para cada plataforma donde se desee ejecutar. En este caso, BEAM fue construido sobre C, así que será tan portable como el programa escrito en C lo permita.
5. La plataforma, en este caso el sistema operativo donde correrá la máquina virtual.

Figura 4.1: Estructura de BEAM VM

La compilación y el intérprete BEAM ya están dados por lo que no será necesario determinar formalizar BEAM o su proceso de compilación. Para la construcción del puerto de este trabajo, será necesario únicamente formalizar Erlang.

4.2. Sintaxis

La sintaxis de Erlang que especificaremos en este trabajo es una versión reducida de la sintaxis real. Utilizaremos Erlang exclusivamente para el puerto y por lo tanto no será necesario ver todas las primitivas que Erlang provee. Si bien la sintaxis de Erlang es importante para trabajar sobre él,

al utilizar la su librería estándar para generar código Erlang solamente será necesario entender su sintaxis abstracta. A continuación definimos de forma informal la sintaxis concreta de Erlang para luego definir la sintaxis abstracta formalmente.

Los programas en Erlang están definidos como módulos, donde cada módulo tiene una secuencia de definiciones de funciones. Cada una de estas funciones se identifica por su nombre y su aridad. Estas funciones pueden exportarse hacia afuera como parte de la interfaz que expone el módulo o pueden también utilizarse de forma interna.

Para especificar el nombre del módulo y las funciones que exporta se especifican atributos del módulo, de la forma $-attr(param_1, \ \ldots \ param_n)$ donde *attr* es el identificador del atributo y $param_i$ son los parámetros del atributo.

La sintaxis de un módulo es simplemente una secuencia de definición de funciones y atributos del módulo que en Erlang se llaman "forms". Cada uno de estos forms finaliza con el carácter "`.`" (punto).

A modo de ejemplo y para generar una intuición básica del manejo de Erlang presentamos un módulo *calculadora* que exporta la función suma.

```
-module(calculadora).

-export([suma/2]).

suma(A, B) ->
  A + B.
```

Nótese que en este código se definen tres forms: los atributos *module* y *export*, y la definición de la función *suma*. El atributo `-export([suma/2]).` exporta la función llamada *suma* con aridad 2.

Las variables en Erlang siempre comienzan con una letra mayúscula, y si bien se puede especificar parámetros con el mismo nombre o incluso con una combinación de estructuras de datos y variables (donde se aplica la unificación para determinar el valor de las variables), para este trabajo esta facilidad no será necesaria. A su vez Erlang permite definir varias veces la misma función (separados utilizando el carácter "`;`" (punto y coma), llamadas "cláusulas"). En estos casos Erlang buscará la primera cláusula de la función que unifique con los parámetros que se enviaron e invocará el cuerpo de esta cláusula.

A modo de ejemplo y para mostrar la unificación presentamos el siguiente código, que es una variación del anterior.

```
-module(calculadora).

-export([suma/2]).

suma(0, B) ->
  B;
suma(A, 0) ->
  A;
suma(A, B) ->
  A + B.
```

Cada una de estas tres cláusulas contiene el nombre de la función (y su aridad implícita), los parámetros y el cuerpo de la cláusula. Este cuerpo de la cláusula está definido por una serie de una o más expresiones, separadas utilizando el carácter "`,`" (coma).

Erlang permite definir varios tipos de expresiones. Las expresiones que utilizaremos en el puerto se presentan a continuación, como parte de la sintaxis abstracta.

4.3. Sintaxis abstracta

La sintaxis de Erlang se puede definir con los siguientes puntos:

- Los programas son secuencias de módulos.
- Los módulos son secuencias de "forms" (definiciones de funciones o atributos).
- Las definiciones de funciones están definidas por el nombre de la función y su aridad, y están compuestas por una o más cláusulas.
- Cada cláusula está definida por los nombres de sus parámetros y el cuerpo de la cláusula.
- El cuerpo de una cláusula está definido por una secuencia de una o más expresiones.

- Una expresión puede ser:
  - Una expresión atómica: números, átomos, strings o lista vacía (`[]`).
  - Una definición de una función anónima.
  - Una aplicación (o llamada) a una función.
  - Una expresión *case* que realiza pattern-matching sobre un valor para determinar la rama que debe evaluar.
  - Una expresión *try-catch* que captura excepciones.

Si bien se pueden definir distintas cláusulas para definir una función, en este trabajo todas las funciones que definiremos tendrán una única cláusula, por lo que no será necesario distinguirlas como una estructura aparte de la definición de una función.

Formalmente, podemos especificar la sintaxis abstracta de la siguiente forma:

$$\begin{array}{lll} \textit{Programas} < p > ::= & < m > < p > & \text{(Módulo seguido de programa)} \\ \quad | & & \text{(Programa vacío)} \end{array}$$

$$\begin{array}{lll} \textit{Módulos} < m > ::= & < d > < m > & \text{(Declaración seguido de módulo)} \\ \quad | & & \text{(Módulo vacío)} \end{array}$$

Declaraciones

$$\begin{array}{lll} < d > ::= & funname(param_1,\ \ldots\ param_n)\ \rightarrow < e_{body} > . & \\ & \text{(Definición de función)} & \\ \quad | & -attrname(< e_1 >,\ \ldots\ < e_n >). & \text{(Atributo)} \end{array}$$

Expresiones

$$
\begin{array}{lll}
< e > ::= & < n > & \text{(Número)} \\
| & < a > & \text{(Átomo)} \\
| & < str > & \text{(String)} \\
| & \texttt{[]} & \text{(Lista vacía)} \\
| & < var > & \text{(Variable)} \\
| & \texttt{fun}\ (param_1,\ \ldots\ param_n)\ \rightarrow < e >\ \texttt{end} & \\
 & \text{(Función anónima)} & \\
| & funname(< e_1 >,\ \ldots\ < e_n >) & \text{(Aplicación)} \\
| & \texttt{try}\ < e >\ \texttt{of}\ var\ \rightarrow < e_{body} > & (\textit{try-catch}) \\
 & \texttt{catch}\ var_{err}\ \rightarrow\ e_{err}\ \texttt{end} & \\
| & \texttt{case}\ < e >\ \texttt{of}\ var\ \rightarrow < e_{body} >\ \texttt{end} & \text{(var } \textit{case}) \\
| & \texttt{case}\ < e >\ \texttt{of}\ < atom - branches >\ \texttt{end} & (\textit{case})
\end{array}
$$

$$
\begin{array}{lll}
< atom{-}branches > ::= & < a > \rightarrow < e_{body} >;\ < atom - branches > & \text{(Rama seguido de r} \\
| & < a > \rightarrow < e_{body} > & \text{(Rama)}
\end{array}
$$

4.4. Semántica operacional

De la misma forma que como definimos relaciones para especificar la semántica operacional de Kλ también definiremos nuevas relaciones y reglas de inferencia para especificar la ejecución de programas en Erlang.

Las relaciones que utilzaremos para definir esta semántica son las siguientes:

- Una relación que determina la *ejecución* de un programa ($\Downarrow_p$).
- Una relación que determina la *carga de funciones globales* de un programa ($\blacktriangleright$).
- Una relación que determina la *evaluación* de una expresión ($\downarrow$).

Para definir esta relaciones primero deberemos definir el concepto de valor y contexto en un programa Erlang.

4.4.1. Valores

En Erlang, de la misma forma que en Shen y Kλ, al evaluar una expresión se obtiene un valor. Este valor es irreducible (canónico), por lo que no se puede realizar más computos sobre el valor resultante para obtener un valor y su representación es única.

Los posibles valores son:

$$
\begin{array}{lll}
\textit{Valores} < v > ::= & < n > & \text{(Número)} \\
\mid & < a > & \text{(Átomo)} \\
\mid & < str > & \text{(String)} \\
\mid & \texttt{[]} & \text{(Lista vacía)} \\
\mid & \langle \texttt{fun}\ (param_1,\ \ldots\ param_n)\ \rightarrow & (\textit{Clausura}) \\
 & \quad < e > & \\
 & \texttt{end},\ \gamma\rangle &
\end{array}
$$

4.4.2. Contexto

El contexto para la semántica en Erlang contiene una asociación de nombres de variables con sus valores correspondientes. Los constructores y las reglas de inferencia para la relación a @! v que definimos sobre este contexto son las mismas que las definidas para la tabla de funciones de Kλ, utilizando asociaciones entre átomos y valores de Erlang.

4.4.3. Tabla de funciones globales

En Erlang las funciones definidas en los módulos tienen acceso global, cualquier parte del programa puede invocarlas, incluso el cuerpo de esa misma función.

Para representar las funciones (a diferencia de Kλ en donde asociamos los nombres de funciones a expresiones que representan la función "currificada") asociamos nombres de funciones a la definición de la función como una expresión de función anónima, con todos sus parámetros. Esto es útil para luego acceder a la aridad y el cuerpo de la función fácilmente.

La relación *@!* que definimos sobre la tabla de funciones es la misma que la relación definida para la tabla de funciones de Kλ pero con distintos elementos que se asocian. En el caso de Erlang se asocia el nombre completo de la función con el módulo como prefijo y su aridad como sufijo (por ejemplo, `calculadora:calculadora/2`) a la definición de función como función anónima. A esta tríada para identificar las funciones globales en Erlang se le llama *MFA* (*module-function-arity*).

4.4.4. Ejecución de programas

En primer lugar partimos de un programa, compuesto por una secuencia de cero o más módulos. Al igual que en Kλ, decimos que un programa ejecuta cuando "dada la tabla que contiene todas las definiciones de función de los módulos del programa y dada una función de inicialización, la misma evalúa a un valor".

Carga de funciones globales

La carga de las funciones globales la definimos a través de una nueva relación ($\blacktriangleright$) que construye la tabla a partir de programas y sus declaraciones.

Esta relación está dada por tres elementos:

- Una tabla de funciones globales f.
- Un programa p, compuesto por una secuencia de módulos.
- Una nueva tabla f' resultante de cargar las nuevas definiciones de función del programa p.

La relación es entonces es la siguiente:

$$\text{load fun form}\ \frac{f \ \longleftarrow\ (mod : fun/n,\ (v_1\ \ldots\ v_n)\ \rightarrow\ e_{body}) \vdash d_2\ \ldots\ d_n\ \blacktriangleright\ f'}{f \vdash fun(v_1\ \ldots\ v_n) \rightarrow\ e_{body}.\ d_2\ \ldots\ d_n\ \blacktriangleright\ f'}$$

$$\text{load nil}\ \frac{}{f\ \vdash \blacktriangleright\ f}$$

Ejecución de funciones de inicialización

En Erlang los módulos están compuestos exclusivamente de funciones y atributos. Al ejecutar un programa en Erlang uno debe especificar la función de partida que realizará la ejecución de todo el programa (de la misma forma que los programas en C o Java tienen una función *main*).

Es por este motivo que Erlang no tiene forma de ejecutar distintas funciones (o expresiones) de inicialización, sino que, al definir un programa se debe indicar su función de inicialización. Representamos la relación de ejecución de un programa como un par que contiene el programa junto con el nombre de su función de inicialización.

Ejecución de un programa

Ahora sí estamos en condiciones de determinar la relación para ejecutar un programa, compuesta por los siguientes elementos:

- Un programa p, compuesto por una secuencia de módulos.
- El identificador de una función de inicialización ($initmod : initfun$).

La notación que utilizaremos es la siguiente:

$$(p, initmod : initfun) \ \Downarrow_p$$

que se lee como "el programa ejecuta con inicialización $initmod : initfun$".

Su definición:

$$\text{exec prog}\ \frac{\epsilon \vdash p \ \blacktriangleright \ f \qquad \langle \epsilon, \ f \rangle \vdash initmod : initfun() \ \downarrow \ S\langle v \rangle}{(p, initmod : initfun) \ \Downarrow_p}$$

4.4.5. Evaluación de expresiones

A continuación presentamos la evaluación de aquellas expresiones que utilizaremos para el puerto.

Evaluación de números, átomos, strings y la lista vacía

Utilizando una sintaxis similar a Shen, Kλ y muchos otros lenguajes de programación, la evaluación de este tipo de expresiones son las propias expresiones atómicas representadas como valores. A cada una de estas reglas la llamaremos: *erl num*, *erl atom*, *erl string* y *erl nil* respectivamente.

Evaluación de variables

Para el caso de Erlang, las variables y los átomos están diferenciados, las variables siempre comienzan con una letra mayúscula.

$$\text{erl var}\ \frac{\gamma\ @!\ v_{var}\ \longrightarrow v}{\langle \gamma,\ f \rangle \vdash v_{var}\ \downarrow\ S\langle v \rangle}$$

Evaluación de `fun`

Una función anónima evalúa a sí misma, representada como valor.

$$\text{erl fun}\ \frac{}{\langle \gamma,\ f \rangle \vdash \texttt{fun}\ (v_1,\ \ldots\ v_n)\ \rightarrow\ e_{body}\ \texttt{end}\ \downarrow\ S\langle \texttt{fun}\ (v_1,\ \ldots\ v_n)\ \rightarrow\ e_{body}\ \texttt{end},\ \gamma \rangle}$$

Evaluación de `case`

Existen dos tipos de estructuras de `case` que vamos a utilizar, aquellos cuyas ramas son del tipo átomo y aquellos que simplemente tienen una única rama del tipo variable.

$$\text{erl case-atom-head}\ \frac{\langle \gamma,\ f \rangle \vdash e\ \downarrow\ S\langle a \rangle \qquad \langle \gamma,\ f \rangle \vdash e_{body}\ \downarrow\ r}{\langle \gamma,\ f \rangle \vdash \texttt{case}\ e\ \texttt{of}\ a\ \rightarrow\ e_{body};\ \overline{rest}\ \texttt{end}\ \downarrow\ r}$$

$$\text{erl case-atom-error}\ \frac{\langle \gamma,\ f \rangle \vdash e\ \downarrow\ E\langle str \rangle}{\langle \gamma,\ f \rangle \vdash \texttt{case}\ e\ \texttt{of}\ a\ \rightarrow\ e_{body};\ \overline{rest}\ \texttt{end}\ \downarrow\ E\langle str \rangle}$$

$$\text{erl case-atom-tail}\ \frac{\langle \gamma,\ f \rangle \vdash e\ \downarrow\ S\langle a \rangle \qquad \langle \gamma,\ f \rangle \vdash \texttt{case}\ e\ \texttt{of}\ \overline{rest}\ \texttt{end}\ \downarrow\ r}{\langle \gamma,\ f \rangle \vdash \texttt{case}\ e\ \texttt{of}\ a_1\ \rightarrow\ e_{body}; \overline{rest}\ \texttt{end}\ \downarrow\ r}\ a\ \neq\ a_1$$

$$\text{erl case-var}\ \frac{\langle \gamma,\ f\rangle \vdash e\ \downarrow\ S\langle v\rangle \qquad \gamma\ \leftarrow\ (var,\ v) \equiv\ \gamma' \qquad \langle \gamma',\ f\rangle \vdash e_{body}\ \downarrow\ r}{\langle \gamma,\ f\rangle \vdash \texttt{case}\ e\ \texttt{of}\ var\ \rightarrow\ e_{body}\ \texttt{end}\ \downarrow\ r}$$

$$\text{erl case-var-error}\ \frac{\langle \gamma,\ f\rangle \vdash e\ \downarrow\ E\langle str\rangle}{\langle \gamma,\ f\rangle \vdash \texttt{case}\ e\ \texttt{of}\ var\ \rightarrow\ e_{body}\ \texttt{end}\ \downarrow\ E\langle str\rangle}$$

Evaluación de `try`

La expresión de `try` se comporta igual que el case del tipo variable para los casos exitosos, pero ejecuta el cuerpo de la cláusula incluida en el `catch` cuando la evaluación de la expresión es un error.

$$\text{erl try-success}\ \frac{\begin{array}{c}\langle \gamma,\ f\rangle \vdash e\ \downarrow\ S\langle v\rangle \qquad \gamma\ \leftarrow\ (var,\ v) \equiv\ \gamma' \\ \langle \gamma',\ f\rangle \vdash e_{body}\ \downarrow\ r\end{array}}{\langle \gamma,\ f\rangle \vdash \texttt{try}\ e\ \texttt{of}\ var\ \rightarrow\ e_{body}\ \texttt{catch}\ var_{err}\ \rightarrow\ e_{err}\ \texttt{end}\ \downarrow\ r}$$

$$\text{erl try-failure}\ \frac{\begin{array}{c}\langle \gamma,\ f\rangle \vdash e\ \downarrow\ E\langle str\rangle \qquad \gamma\ \leftarrow\ (var_{err},\ str) \equiv\ \gamma' \\ \langle \gamma',\ f\rangle \vdash e_{err}\ \downarrow\ r\end{array}}{\langle \gamma,\ f\rangle \vdash \texttt{try}\ e\ \texttt{of}\ var\ \rightarrow\ e_{body}\ \texttt{catch}\ var_{err}\ \rightarrow\ e_{err}\ \texttt{end}\ \downarrow\ r}$$

4.4.6. Evaluación de aplicación de funciones

La aplicación de funciones en Erlang se hace invocando la función con cero o más parámetros. Esta función puede ser una función anónima o una global. Al realizar cualquiera de estas aplicaciones de función, el número de parámetros de la llamada debe coincidir con la aridad de la función.

Evaluación de aplicación de funciones anónimas

En este caso, el operador evalúa a la función anónima de aridad n y contexto de clausura γ' y se sustituye cada uno de los valores de las n expresiones evaluadas en un nuevo contexto (γ''). Con este nuevo contexto, se evalúa el cuerpo de la función.

Si el operador o cualquiera de los operandos evalúa a un error, el mismo se retorna como evaluación de toda la expresión de aplicación (evaluando primero el operador y luego los operandos de izquierda a derecha). Para estos casos creamos una notación $\langle \gamma,\ f \rangle \vdash \overline{e_i} \ \downarrow \ r$ que se interpreta como:

- Si todas las expresiones evalúan a un valor exitoso, entonces $r \ \equiv \ \overline{v_i}$ donde $\overline{v_i}$ es la lista de los valores resultantes de evaluar cada una de la expresiones y v_i es el valor resultante de evaluar e_i.
- Si una de las expresiones evalúa a un error, entonces $r \ \equiv \ E\langle str \rangle$ donde $E\langle str \rangle$ es el resultado erróneo de la primer expresión que evalúa a un error.

$$\text{erl app-anon}\ \frac{\langle \gamma,\ f \rangle \vdash e_{fun} \ \downarrow \ S\langle\langle \texttt{fun}\ (var_1\ \ldots\ var_n) \rightarrow\ e_{body}\ \texttt{end},\ \gamma' \rangle\rangle \qquad \gamma' \longleftarrow \overline{(var_i,\ v_i)} \ \equiv \ \gamma'' \qquad \langle \gamma,\ f \rangle \vdash \overline{e_i} \ \downarrow \ S\langle \overline{v_i} \rangle \qquad \langle \gamma'',\ f \rangle \vdash e_{body} \ \downarrow \ r}{\langle \gamma,\ f \rangle \vdash e_{fun}(e_1,\ \ldots, e_n) \ \downarrow \ r}$$

$$\text{erl app-error-efun}\ \frac{\langle \gamma,\ f \rangle \vdash e_{fun} \ \downarrow \ E\langle str \rangle}{\langle \gamma,\ f \rangle \vdash e_{fun}(e_1,\ \ldots, e_n) \ \downarrow \ E\langle str \rangle}$$

$$\text{erl app-error-params}\ \frac{\langle \gamma,\ f \rangle \vdash e_{fun} \ \downarrow \ S\langle\langle \texttt{fun}\ (var_1\ \ldots\ var_n) \rightarrow\ e_{body}\ \texttt{end},\ \gamma' \rangle\rangle \qquad \langle \gamma,\ f \rangle \vdash \overline{e_i} \ \downarrow \ E\langle str \rangle}{\langle \gamma,\ f \rangle \vdash e_{fun}(e_1,\ \ldots, e_n) \ \downarrow \ E\langle str \rangle}$$

Evaluación de aplicación de funciones globales

Las funciones globales están identificadas el nombre del módulo, el nombre de la función y su aridad. Para aplicar una función global es suficiente con especificar el nombre del módulo y de la función. La aridad se infiere a través del número de argumentos de la aplicación.

De la misma forma que con la aplicación de funciones anónimas, si uno de los parámetros de la función evalúa a un error (evaluando de izquierda a derecha), toda la aplicación evalúa a un error.

$$\text{erl app-global-fun}\ \frac{f\ @!\ (mod : fun/n) \ \longrightarrow e_{fun} \qquad \langle \gamma,\ f \rangle \vdash e_{fun}(e_1\ \ldots\ e_n) \ \downarrow \ r}{\langle \gamma,\ f \rangle \vdash mod : fun(e_1\ \ldots\ e_n) \ \downarrow \ r}$$

4.4.7. Atributos de módulo

Como ya mencionamos, los módulos están compuestos por definiciones de funciones y atributos de módulo. Los atributos que utilizaremos son los siguientes:

- `module` – Especifica el nombre del módulo. Este atributo es obligatorio incluir en todos los módulos y sólo se debe definir una vez por cada módulo (por ejemplo, `-module(calculadora)`).
- `export` – Especifica una o más funciones a exportar. Cada función deberá ser especificada con su nombre y su aridad (por ejemplo, `-export([suma/2, abs/1])`).

Capítulo 5

Construcción del puerto

Teniendo ahora definida la sintaxis y semántica del lenguaje de origen (Kλ) y la sintaxis (abstracta) y semántica del lenguaje de destino (Erlang) estamos en condiciones de construir el puerto.

La construcción del puerto involucra los siguientes pasos:

1. Definir lexer y parser para obtener las distintas expresiones a partir de programas en Shen.
2. Definir la generación de código: cómo transformar programas Kλ en Erlang, preservando la semántica.

Las siguientes secciones definen cada uno de estos pasos.

5.1. Compilación – Lexing y parsing

La primera parte del proceso de compilación consiste en el *lexing* y *parsing* de Kλ, a través de la utilización de las herramientas que Erlang provee para el *lexing* y *parsing*.

5.1.1. Lexing

La sintaxis de Kλ está compuesta por *S-expressions* y literales. Donde estos literales pueden ser números, símbolos, cadenas de caracteres (delimi-

tados por comillas dobles) o la lista vacía `()`.

La herramienta utilizada para extraer los tokens de un archivo es el *lexer* incluído por defecto en la librería estándar de Erlang: *Leex*. Especificar *tokens* con Leex es muy similar a hacerlo con Lex, una herramienta utilizada para realizar el proceso de *lexing* en el lenguaje C. Definir tokens en Leex se debe definir las definiciones y las reglas.

Las definiciones utilizadas son las siguientes:

- S - Un guión opcional para permitir números con signo.
- D - Dígitos del 0 al 9.
- WS - Caracteres a ignorar (*white space*).
- A - Caracteres de los átomos: cualquier carácter menos comillas dobles, paréntesis y *white space*.

```
S  = -?
D  = [0-9]
WS = [\000-\s]
A  = [^()\"\000-\s]
```

Las reglas utilizadas para extraer los tokens del texto son las siguientes:

```
%% Numbers
{S}{D}+       : {token, {number, TokenLine, list_to_integer(TokenChars)}}.
{S}{D}+\.{D}+ : {token, {number, TokenLine, list_to_float(TokenChars)}}.

%% String
\"[^\"]*\"    : {token, {string,
                         TokenLine,
                         string:substr(TokenChars, 2, TokenLen - 2)}}.

%% Symbols
{A}+          : {token, {symbol, TokenLine, list_to_atom(TokenChars)}}.

%% Parentheses
[()]          : {token, {list_to_atom(TokenChars), TokenLine}}.

%% Whitespace/comments
{WS}+         : skip_token.
```

5.1.2. Parsing

Realizar el mecanismo de *parsing* para extraer las expresiones de Kλ es suficiente con obtener todos sus literales y *S-expressions* que contienen únicamente más expresiones.

La herramienta utilizada para realizar el *parsing* sobre los tokens es el *parser* incluído por defecto la librería estándar de Erlang: Yeec. Especificar gramáticas con Yeec es también muy similar a como se hace con YACC, una herramienta utilizada para realizar el proceso de *parsing* en el lenguaje C. Definir una gramática consta de dos secciones: las definiciones y las reglas.

Antes de definir las definiciones se deben listar todas las categorías de terminales y no-terminales de la gramática, junto con la categoría que se utilizará como el punto de partida (en este caso el token *expr_list*). Las categorías a utilizar son los siguientes:

- Terminales - Estos son los tokens extraídos como salida del proceso del lexer
  - *symbol*
  - *number*
  - *string*
  - `(` y `)`
- No-terminales
- *expr* - Compuesto por cualquiera de las siguientes categorías: *list*, *number*, *string* y *symbol*.
- *expr_list* - Compuesto por una secuencia de expresiones.
- *list* - Compuesto por una secuencia de 0 o más expresiones encerrado por paréntesis.

La gramática entonces se especifica de la siguiente forma:

```
Nonterminals
  expr_list expr list.

Terminals
```

```
  symbol number string '(' ')'.

Rootsymbol expr_list.

expr_list -> expr expr_list : ['$1' | '$2'].
expr_list -> expr : ['$1'].

expr -> list : '$1'.
expr -> number : '$1'.
expr -> string : '$1'.
expr -> symbol : '$1'.

list -> '(' ')' : [].
list -> '(' expr_list ')' : '$2'.
```

5.2. Compilación – Generación de código

La compilación de Kλ a Erlang consiste en la compilación de programas en Kλ (como una secuencia de definiciones) a módulos Erlang. Para definir el pasaje de un lenguaje a otro utilizaremos exclusivamente la sintaxis abstracta. Esto no sólo es con el fin de simplificar la generación del código (el pasaje de la sintaxis abstracta a la sintaxis real puede realizarse fácilmente luego), sino que para compilar a programas en Erlang su utilizó la librería estándar de Erlang (*erl_syntax*) que construye términos en Erlang a partir de funciones. Es decir, no necesitaremos generar los programas en Erlang como texto.

Luego de generar los módulos en Erlang uno puede luego compilar a BEAM y evaluarlos; de esta forma es que realizaremos el compilador e intérprete.

A su vez, como mencionamos anteriormente, cuando en Kλ se habla de "primitivas", se refiere tanto a las "formas especiales" que las expresiones de Kλ pueden tener (por ejemplo, `if`, `and`, `freeze`) como a las funciones predefinidas que hay que implementar para el puerto. Compilar o interpretar Kλ, es contemplar las 57 primitivas definidas por el lenguaje vistas anteriormente. Para la formalización que realizaremos a continuación sólo utilizaremos las formas especiales, ya que las funciones que hay que proveer para completar el puerto (por ejemplo, *open*, *read-byte*) son triviales de implementar y requieren definir nuevas estructuras complejas para su formalización (como la

interacción con el sistema operativo o la definición de tipos de expresiones) que no aportan para definir Kλ. La implementación de estas funciones se especificará luego de forma informal en el Apéndice B.

5.2.1. Compilación de programas

Compilar programas de Kλ, compuesto por una secuencia de definiciones de funciones y expresiones de primer nivel, implica generar nuevos módulos Erlang que contienen definiciones de función y atributos.

Para compilar definiciones de funciones definimos un nuevo módulo con una única función. El nombre de este módulo y de la función son generados a partir del nombre de la función de Kλ y el número de parámetros que recibe son los mismos que los utilizados en la definición. Por ejemplo:

```
(defun suma (A B)
  (+ A B))
```

compilará al módulo

```
-module(suma).
-export([suma/2]).

suma(A, B) ->
  '+':'+'(A, B).
```

Para compilar expresiones de primer nivel también definimos un nuevo módulo con una única función (llamada `tle`). El nombre del módulo es único y aleatorio y la función definida dentro de él deberá ser llamada al momento de cargar el módulo.

```
(set counter (+ (value counter) 1))
```

compilará al módulo

```
-module(tle_46db3b3101476f60).
-export([tle/0]).

tle() ->
  'set':'set'(counter, '+':'+'('value':'value'(counter), 1)).
```

Todos los módulos que contienen las primitivas de Kλ o que son parte de la librería estándar de Shen deberán ser cargados antes de ejecutar un programa.

5.2.2. Compilación de expresiones

Esta función, que la llamaremos *compile*, es la encargada de generar expresiones Erlang a partir de expresiones Kλ. Uno de los objetivos principales de este trabajo es definir y probar la corrección de esta función.

Ambiente (Env)

Para definir *compile* precisamos una nueva estructura auxiliar: el ambiente (o *Env*) que representa el conjunto de variables locales accesibles desde determinado punto del código.

De la misma forma que definimos el contexto en Kλ y Erlang, también haremos uso de dos constructores para construir el ambiente:

- Ambiente vacío ($\epsilon : Env$) que retorna un ambiente sin ninguna variable.
- Añadir variable ($\longleftarrow : [Env \times < s >] \rightarrow Env$), que retorna el nuevo ambiente con la nueva variable s (símbolo) añadida.

Este contexto representa una estructura de datos sobre la que se pueden definir funciones para determinar si una variable está presente en el ambiente o no. En este caso será necesario definir una única función *exists*, que dado un ambiente y un símbolo s devuelve `true` si el símbolo está dentro del ambiente y `false` si no. La definición de *exists* es la siguiente:

$$exists \ : \ Env \times \ < s > \rightarrow \texttt{true} \mid \texttt{false}$$

$$\begin{array}{lll} exists(\epsilon,\ s) & = \texttt{false} & \\ exists(\Sigma \ \longleftarrow \ s,\ s) & = \texttt{true} & \\ exists(\Sigma \ \longleftarrow \ s_1,\ s_2) & = exists(\Sigma,\ s_2) & (s_1 \neq s_2) \end{array}$$

Funciones auxiliares

Previo a definir la generación de código también definiremos las siguientes funciones, que utilizaremos para definir *compile*:

- *n2n* – Dado un número en Kλ retorna su representación como un número en Erlang.
- *str2str* – Dado un string en Kλ retorna su representación como un string en Erlang.
- *s2a* – Dado un símbolo en Kλ retorna su representación como un átomo en Erlang.
- *s2var* - Dado un símbolo en Kλ retorna su representación como una variable en Erlang. Como las variables en Erlang siempre comienzan con mayúsculas, esto se logra simplemente con agregar un carácter (en este caso elegimos utilizar *"V"*) como prefijo de la representación textual del símbolo.
- *s2funname* - Dado un símbolo en Kλ retorna su representación como el operador de una llamada global (por ejemplo, $s2funname(suma) \equiv suma : suma$).
- *newvar* - Dado un ambiente retorna un nuevo nombre de variable único, no utilizado en todo el programa.

compile

Dado el ambiente y la forma de construir expresiones en Kλ y Erlang ahora estamos en condiciones de definir *compile*:

$$compile \ : < e_{kl} > \ \times Env \ \rightarrow < e_{erl} >$$

Para cada forma posible de expresión en Kλ esta función retorna una expresión en Erlang. A continuación definimos *compile* para cada posible forma de expresión en Kλ.

Compilación de números, strings y lista vacía

Las expresiones atómicas en Kλ están compuestas por números, strings, símbolos y la lista vacía. Para cada uno de estos tipos de expresiones existe una expresión en Erlang que también evalúa a su valor.

$$
\begin{array}{lll}
compile(n,\ \Sigma) & = n2n(n) & \text{(Número)} \\
compile(str,\ \Sigma) & = str2str(str) & \text{(String)} \\
compile(\texttt{()},\ \Sigma) & = [] & \text{(Lista vacía)}
\end{array}
$$

Compilación de símbolos y variables

Para el caso de compilar símbolos, como vimos en la semántica de Kλ, tienen un significado distinto de evaluación dependiendo si los mismos están o no definidos (como variables) en el contexto. Es por este motivo que tenemos dos casos que considerar al compilar un símbolo:

$$
\begin{array}{llll}
compile(s,\ \Sigma) & = s2a(s) & (si\ exists(\Sigma,\ s) = \texttt{false}) & \text{(Símbolo a átomo)} \\
compile(s,\ \Sigma) & = s2var(s) & (si\ exists(\Sigma,\ s) = \texttt{true}) & \text{(Símbolo a var)}
\end{array}
$$

Compilación de expresiones `if`

Las expresiones `if` en Kλ se pueden especificar directamente como una estructura de `case` de Erlang, que contempla para la condición los dos valores booleanos y evalúa la expresión e_{true} o e_{false} compilada.

$$
\begin{array}{l}
compile((\texttt{if}\ e\ e_{true}\ e_{false}),\ \Sigma) = \\
\quad \texttt{case}\ compile(e,\ \Sigma)\ \texttt{of} \\
\quad\quad \texttt{true}\ \rightarrow\ compile(e_{true}, \Sigma)\texttt{;} \\
\quad\quad \texttt{false}\ \rightarrow\ compile(e_{false}, \Sigma) \\
\quad \texttt{end}
\end{array}
$$

Compilación de expresiones `cond`

Las expresiones `cond` en Kλ se pueden especificar directamente como una estructura de `case` anidados de Erlang. En el caso de que una condición evalúe a `true` se retorna el cuerpo compilado de esa pareja condición-cuerpo. En el caso de que una condición evalúe a `false`, se evalúa el siguiente `case` con el resto de las parejas condición-cuerpo.

$$
\begin{array}{l}
\mathit{compile}((\texttt{cond}\ (e\ e_{body})\ \overline{rest}),\ \Sigma) = \\
\quad \texttt{case}\ \mathit{compile}(e,\ \Sigma)\ \texttt{of} \\
\quad\quad \texttt{true}\ \rightarrow\ \mathit{compile}(e_{body},\ \Sigma); \\
\quad\quad \texttt{false}\ \rightarrow\ \mathit{compile}((\texttt{cond}\ \overline{rest}),\ \Sigma) \\
\quad \texttt{end}
\end{array}
$$

El caso base se aplica cuando la estructura `cond` tiene una sóla condición. En este caso, sólo se maneja cuando la expresión evalúa a `true`.

$$
\begin{array}{l}
\mathit{compile}((\texttt{cond}\ (e\ e_{body})),\ \Sigma) = \\
\quad \texttt{case}\ \mathit{compile}(e,\ \Sigma)\ \texttt{of} \\
\quad\quad \texttt{true}\ \rightarrow\ \mathit{compile}(e_{body},\ \Sigma); \\
\quad \texttt{end}
\end{array}
$$

Compilación de expresiones `trap-error`

Las expresiones `trap-error` en Kλ se pueden especificar como una estructura `try` de Erlang.

$$
\begin{array}{l}
\mathit{compile}((\texttt{trap-error}\ e\ e_{handler}),\ \Sigma) = \\
\quad \texttt{try}\ \mathit{compile}(e,\ \Sigma)\ \texttt{of} \\
\quad\quad s2var(sv)\ \rightarrow\ s2var(sv) \\
\quad \texttt{catch} \\
\quad\quad s2var(serr)\ \rightarrow\ \mathit{compile}((e_{handler}\ serr),\ \Sigma \longleftarrow serr) \\
\quad \texttt{end}
\end{array}
$$

donde *sv* y *serr* son variables nuevas que se obtienen a través de la función *newvar*.

Compilación de expresiones `and` y `or`

Las expresiones `and` y `or` en Kλ se pueden especificar como una estructura de `case` en Erlang.

$$
\begin{array}{l}
compile((\texttt{and}\ e_1\ e_2),\ \Sigma) = \\
\quad \texttt{case}\ compile(e_1,\ \Sigma)\ \texttt{of} \\
\quad\quad \texttt{true}\ \rightarrow\ compile(e_2,\ \Sigma); \\
\quad\quad \texttt{false}\ \rightarrow\ \texttt{false} \\
\quad \texttt{end}
\end{array}
$$

$$
\begin{array}{l}
compile((\texttt{or}\ e_1\ e_2),\ \Sigma) = \\
\quad \texttt{case}\ compile(e_1,\ \Sigma)\ \texttt{of} \\
\quad\quad \texttt{false}\ \rightarrow\ compile(e_2,\ \Sigma); \\
\quad\quad \texttt{true}\ \rightarrow\ \texttt{true} \\
\quad \texttt{end}
\end{array}
$$

Compilación de expresiones `lambda`

Las expresiones `lambda` en Kλ se pueden especificar como una función de un sólo parámetro en Erlang.

$$
\begin{array}{l}
compile((\texttt{lambda}\ s\ e_{body}),\ \Sigma) = \\
\quad \texttt{fun}\ (s2var(s))\ \rightarrow\ compile(e_{body},\ \Sigma\ \leftarrow\ s)\ \texttt{end}
\end{array}
$$

En este caso, la variable de la función se agrega al ambiente Σ para que dentro del cuerpo de la función el símbolo s se interprete como variable.

Compilación de expresiones `let`

Esta estructura asigna un valor a una variable y ejecuta el cuerpo del `let`, por lo que será necesario agregar el símbolo de la variable (s) al ambiente para compilar el cuerpo.

$$
\begin{aligned}
&compile((\texttt{let}\ s_{var}\ e\ e_{body}),\ \Sigma) = \\
&\quad \texttt{case}\ compile(e,\ \Sigma)\ \texttt{of} \\
&\qquad s2var(s)\ \rightarrow\ compile(e_{body},\ \Sigma\ \leftarrow\ s) \\
&\quad \texttt{end}
\end{aligned}
$$

Compilación de "valores congelados"

Estas expresiones compilan a una función anónima de Erlang sin parámetros para luego ser llamada y evaluada.

$$compile((\texttt{freeze}\ e_{body}),\ \Sigma) = \texttt{fun}\ ()\ \rightarrow\ compile(e_{body},\ \Sigma)\ \texttt{end}$$

5.2.2.1. Compilación de aplicación de funciones

Como lo especifica la semántica de Kλ, toda "*S-expression*" que no tenga ninguna de las estructuras mencionadas anteriormente se considera una aplicación de función. Estas aplicaciones pueden ser de dos tipos dependiendo del operador, cuando el operador es un símbolo (y no es una variable) entonces se toma como una aplicación de una función global, de lo contrario, se toma como una aplicación de una función lambda o expresión "congelada".

Compilación de aplicación de funciones globales

La evaluación de la aplicación de funciones globales en Erlang difere de la aplicación en Kλ, porque en Kλ tanto las funciones anónimas como las funciones globales están "currificadas", es decir, siempre toman un único parámetro, mientras que en Erlang pueden tomar más de uno.

La compilación a Erlang se podría haber diseñado de esta misma forma, asegurando que las funciones globales estuviesen "currificadas". Esto se lograría si todas las funciones globales fuesen funciones de un único parámetro que retornan el resto de la función currificada.

De todas formas, por eficiencia y legibilidad del código generado (y por lo tanto mejor entendimiento de los errores en tiempo de ejecución) se compiló las definiciones de función como funciones globales de Erlang que toman tantos parámetros como parámetros tiene esa definición. Por ejemplo, `(defun`

`suma (a b) ...)` se compiló como una definición de función de Erlang identificada como `suma:suma/2`.

Entonces, compilar aplicaciones de funciones globales de Kλ a Erlang no resulta directo, ya que las aplicaciones de función en Erlang no están currificadas: deben recibir todos sus parámetros al ser aplicadas.

Esto provoca una potencial diferencia entre el número de parámetros que la función definida recibe y el número de parámetros que recibe en su aplicación. Siguiendo la semántica de Kλ las funciones globales pueden recibir cualquier número de parámetros, por lo que tendremos que hacer una distinción entre tres casos: cuando el número de parámetros de la aplicación es menor, igual o mayor al número de parámetros que acepta la función.

En el primer caso, cuando el número de parámetros de la aplicación es menor, se deberá retornar funciones anónimas de un parámetro anidadas, que reciben el resto de los parámetros de forma individual para luego realizar la llamada a la función global de Erlang.

$$compile((s\ e_1\ \ldots\ e_n),\ \Sigma) = \\ \texttt{fun}\ (s2var(sv))\ \rightarrow\ compile((s\ e_1\ \ldots\ e_n\ sv),\ \Sigma\ \longleftarrow\ sv)\ \texttt{end}$$

Donde sv es una variable nueva que se obtiene a través de la función $newvar$ ($sv = newvar(\Sigma)$).

En el segundo caso, cuando el número de parámetros de la aplicación es igual, simplemente se aplica la función global con esos parámetros.

$$compile((s\ e_1\ \ldots\ e_n),\ \Sigma) = \\ s2funname(s)(compile(e_1,\ \Sigma)\ \ldots\ compile(e_n,\ \Sigma)))$$

En el tercer caso, cuando el número de parámetros de la aplicación es mayor, primero se realiza la llamada a la función global con los m parámetros que recibe y luego se realiza la aplicación del resto de los parámetros de a uno (como si fuese una aplicación de funciones lambda).

Para este caso utilizaremos dos nuevas funciones auxiliares:

- *take* - Recibe un número m y una lista y retorna los primeros m elementos de la lista.

- *skip* - Recibe un número m y una lista y retorna la lista sin los primeros m elementos.

$$compile((s\ e_1\ \ldots\ e_n),\ \Sigma) = \\ compile(((s\ take(m,\ e_1,\ \ldots\ e_n))\ skip(m,\ e_1,\ \ldots\ e_n)))$$

Aquí vemos cómo compilar la aplicación de una función que recibe m parámetros pero se aplica con un número mayor. En este caso se aplica la función con los primeros m parámetros – $(s\ take(m,\ e_1,\ \ldots\ e_n))$ (llamémosle e_{fun}) – y el resultado luego se aplica con el resto de los parámetros – $(e_{fun}\ skip(m,\ e_1,\ \ldots\ e_n))$.

Compilación de aplicación de funciones lambda

Cuando el operador no resulta ser un símbolo (una función global) o una variable entonces la aplicación de la función es simplemente la aplicación de funciones anónimas en Erlang. En el caso que la expresión tenga uno o más parámetros entonces el operando se aplica al primer argumento y de forma recursiva se continúa aplicando al resto.

$$compile((e_1\ e_2),\ \Sigma) = compile(e_1,\ \Sigma)(compile(e_2,\ \Sigma))$$
$$compile((e_1\ e_2\ \overline{e_{rest}}),\ \Sigma) = compile(((e_1\ e_2)\ \overline{e_{rest}}),\ \Sigma)$$

Compilación de aplicación de "valores congelados"

Cuando el operando no resulta ser una función global y la aplicación no recibe ningún parámetro entonces consideramos el operando como un "valor congelado". En este caso, compilar esa expresión es aplicar el operando compilado sin ningún parámetro.

$$compile((e),\ \Sigma) = compile(e,\ \Sigma)()$$

5.3. Implementación de funciones y símbolos globales

La forma en la que se evalúan las primitivas que son funciones y símbolos globales de Kλ están especificados de manera informal y se deja (parcialmente) liberado a la implementación de su puerto. Sin embargo, el comportamiento de las mismas en cuanto a lo que reciben y retornan está formalmente establecido dentro de la especificación de Kλ online [7].

Dentro del Apéndice B incluímos la lista completa de las funciones a implementar en Kλ como primitivas y la forma en la que fueron implementadas para este puerto. En el caso de la función *eval-kl*, sin embargo, vale la pena aclarar su funcionamiento.

5.3.1. Función global: eval-kl

eval-kl recibe una lista (estructurada en Kλ como pares anidados) y evalúa una definición de Kλ. En el caso de que la declaración sea una definición de función, se deberá agregar la misma como una función global que podrá luego ser aplicada. En el caso que sea una expresión de primer nivel, se deberá evaluar y retornar el valor resultante.

Para lograr esto realizamos los siguientes pasos, utilizando funcionalidades provistas por la librería estándar de Erlang:

1. Compilamos el código utilizando la función *compile* del puerto.
2. Generamos el código BEAM utilizando `compile:forms/1`. En cualquiera de los casos, compila la declaración de Kλ que debería generar un único *form* de Erlang.
3. Evaluamos el código BEAM utilizando `code:load_binary/3` que recibe el código BEAM de un módulo, lo evalúa y lo carga dentro de los módulos globales disponibles.
4. Si el código Kλ original era una expresión de primer nivel, se llama a Mod : `tle`().

De esta forma se cargan en tiempo de ejecución nuevos módulos erlang y se evalúan las expresiones de primer nivel.

5.4. Corrección de la generación de código

La formalización de la generación del código requiere definir las estructuras y relaciones (algunas de ellas funciones) necesarias para luego probar la corrección y completud de la generación de código de Kλ a Erlang.

En particular, vamos a probar la corrección y completud de la generación de expresiones, ya que estas (las expresiones de primer nivel) son las que se evalúan al ejecutar un programa.

Para cada forma posible de expresión en Kλ *compile* retorna una expresión en Erlang. Para todas y cada una de estas formas probaremos que la función es correcta (corrección) tomando como referencia la semántica de ambos lenguajes para probar que existe trazabilidad.

Para realizar estas pruebas primero debemos definir el resto de las funciones y estructuras que forman parte de la formalización para realizar las pruebas.

5.4.1. Función de conversión de valores (vconv)

Para evaluar expresiones en Kλ definimos una relación que especificaba el resultado de evaluar esa expresión; lo mismo hicimos para las expresiones en Erlang. En ambos casos los resultados tenían la forma $S\langle v\rangle$ o $E\langle str\rangle$. Esta función convierte un resultado de Kλ a Erlang.

$$vconv \ : < v_{kl} > \rightarrow < v_{erl} >$$

Su definición:

$$
\begin{array}{lll}
vconv(n) & = n2n(n) & \text{(Número)} \\
vconv(s) & = s2a(s) & \text{(Símbolo)} \\
vconv(str) & = str2str(str) & \text{(String)} \\
vconv(\texttt{()}) & = [] & \text{(Lista vacía)} \\
vconv(\{(\texttt{lambda}\ s_{param}\ e_{body}), \gamma\}) & = & \text{(Clausura)} \\
\quad \{\texttt{fun}\ (s2var(s_{param})) \rightarrow & & \\
\quad\quad compile(e_{body},\ vars(\gamma) \longleftarrow s_{param}) & & \\
\quad \texttt{end},\ cconv(\gamma)\} & & \\
vconv(\{(\texttt{freeze}e_{body}), \gamma\}) & = & \text{(Valor “congelado”)} \\
\quad \{\texttt{fun}\ () \rightarrow & & \\
\quad\quad comp(e_{body},\ vars(\gamma)) & & \\
\quad \texttt{end}, cconv(\gamma)\} & &
\end{array}
$$

Nótese que para definir la conversión utilizamos algunas funciones adicionales. En el caso de las funciones *n2n*, *s2a*, *str2str* simplemente convierten los números, símbolos (a átomos) y strings a su representación correspondiente en Erlang.

Para definir la clausura utilizamos además una función adicional: *s2var* que convierte un símbolo a una variable válida en Erlang. En este caso, como las variables en Erlang comienzan con mayúsculas, simplemente se agrega el prefijo “V” al símbolo, de esta forma se preserva las referencias a las variables al compilar.

En el caso de la conversión de los valores `lambda` y `freeze`, para lograr compilar el cuerpo de la función necesitamos una nueva función *vars*, que toma el contexto γ y construye un ambiente que contiene todas las variables definidas en γ, incluyendo la variable definida como parámetro de la definición de `lambda`. Esto hace que el ambiente Σ utilizado para la compilación siempre tenga las variables de clausura de la función.

5.4.2. Función de conversión de contextos (cconv)

La semántica de ambos lenguajes la definimos teniendo en cuenta un “contexto”. En ambos casos también el contexto contiene una asociación de variables (símbolos en Kλ y nombres de variable en Erlang) a valores. Una variable siempre evalúa a su valor en el contexto.

Para poder realizar una conversión de contextos entre lenguajes definimos *cconv*, que convierte cada una de las asociaciones de símbolos y valores de

Kλ a nombres de variable y valores de Erlang, utilizando *vconv*.

$$cconv \ : < \gamma_{kl} > \rightarrow < \gamma_{erl} >$$

Su definición:

$$\begin{array}{lll} cconv(\epsilon) & = \epsilon & \text{(Contexto vacío)} \\ cconv(\gamma \longleftarrow (s,\ v)) & = cconv(\gamma) \ \longleftarrow \ (s2a(s),\ vconv(v)) & \text{(Contexto con asociación)} \end{array}$$

5.4.3. Función de conversión de tabla de funciones (fconv)

La tabla de funciones en Kλ la definimos como una asociación entre símbolos y expresiones. En Erlang, la tabla de funciones es una asociación entre la tríada *mod : fun / arity* (llamada "*MFA*") y expresiones. Para convertir de la tabla de funciones de Kλ a Erlang, vamos a precisar una forma de pasar de un símbolo a una tríada de la forma *MFA*. Para ello, vamos a utilizar el símbolo como el nombre del módulo y como el nombre de la función. De esta forma, el símbolo s será utilizado como $s2a(s) : s2a(s)/arity$.

$$fconv \ : < f_{kl} > \rightarrow < f_{erl} >$$

Su definición:

$$\begin{array}{lll} fconv(\epsilon) & = \epsilon & \text{(Tabla vacía} \\ fconv(f \longleftarrow (s,\ e_{kl})) & = & \text{(Tabla con asociación)} \\ \quad fconv(f) \ \longleftarrow & & \\ \qquad (s2a(s) : s2a(s)/arity(e_{kl}), & & \\ \qquad decurrify(compile(e_{kl},\ \epsilon))) & & \end{array}$$

Para esta definición introducimos dos nuevas funciones:

- $arity : e_{kl} \ \rightarrow \ \mathbb{N}$ - Dado una expresión Kλ de cero o más expresiones lambda determina su aridad basado en el número de expresiones lambda anidadas.

- *decurrify* : $e_{erl} \rightarrow e_{erl}$ - Dado una expresión Erlang de n funciones anónimas anidadas construye una única función anónima que recibe n parámetros.

5.4.4. Función de conversión de resultado (rconv)

Para evaluar expresiones en Erlang y Kλ definimos relaciones que especifican el resultado de evaluarlas. En ambas relaciones los resultados tenían la forma $S\langle v\rangle$ o $E\langle str\rangle$. Esta función convierte un resultado de Kλ a Erlang. En el caso exitoso el valor devuelto es el propio valor convertido a Erlang utilizando *vconv*.

$$rconv : < r_{kl} > \rightarrow < r_{erl} >$$

Su definición:

$$\begin{array}{lll} rconv(S\langle v\rangle) & = S\langle vconv(v)\rangle & \text{(Resultado exitoso)} \\ rconv(E\langle str\rangle) & = E\langle str2str(str)\rangle & \text{(Resultado erróneo)} \end{array}$$

5.4.5. Corrección de *compile*

Dadas las funciones definidas anteriormente ahora estamos en condiciones de probar la corrección de la función *compile*. Para ello, debemos definir qué significa que la función *compile* sea correcta.

A través de la semántica operacional de Kλ y Erlang logramos determinar cómo se evalúan las expresiones en cada lenguaje. Al compilar de Kλ a Erlang entonces debemos preservar la forma en que se evalúan sus expresiones. Esto es lo que llamamos "trazabilidad". Un programa en Kλ compilado a cualquier otro lenguaje debe respetar la semántica de Kλ. Probar la trazabilidad del proceso de compilación es probar la corrección del compilador.

Las expresiones en Kλ y Erlang evalúan bajo contextos distintos a resultados distintos: por un lado están los contextos, tablas de funciones y resultados de Kλ y por otro lado están los contextos, tablas de funciones y resultados de Erlang. Para determinar una equivalencia de contextos, tablas de funciones y resultados al pasar de Kλ a Erlang definimos las funciones

cconv, *fconv* y *rconv* que convierten el contexto, las tablas de funciones y los resultados de la semántica de Kλ a la semántica de Erlang respectivamente.

La función *compile* entonces es correcta cuando se cumple la siguiente proposición:

$$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e_{kl} \downarrow r_{kl} \rightarrow \langle cconv(\gamma_{kl}),\ fconv(f_{kl}) \rangle \vdash compile(e_{kl},\ vars(\gamma_{kl})) \downarrow rconv(r_{kl})$$

que se lee como "para todo e_{kl} que evalúa a un resultado r_{kl} bajo el contexto γ_{kl} y tabla de funciones f_{kl}, $compile(e_{kl},\ vars(\gamma_{kl}))$ debe evaluar al resultado $rconv(r_{kl})$ bajo el contexto $cconv(\gamma_{kl})$ y tabla de funciones $fconv(f_{kl})$".

Las pruebas se realizarán de manera informal (narrativa) pero siempre utilizando las estructuras ya definidas (relaciones, funciones y igualdades) para determinar cómo se logra llegar al resultado. Las mismas se encuentran en el Apéndice A de este documento.

5.5. Extendiendo Shen

Para lograr una mejor integración con los programas en Erlang introducimos tres nuevas funciones globales (primitivas) para el manejo de mensajes entre procesos Erlang.

Los procesos en Erlang, identificados por un *pid*, tienen una "casilla de mensajes" (*mailbox*) que contiene todos los mensajes aún no procesados que fueron enviados a ese proceso. Para enviar un mensaje desde un proceso a otro se utiliza el operador *!* con el *pid* y el mensaje a enviar. Por ejemplo, para enviar el mensaje formado por el átomo `cheese` al *pid* registrado con el nombre `cheese_truck` se escribe `cheese_truck ! cheese`. Esta primitiva la llamaremos `erl.send`, y la implementamos de la siguiente forma:

```
-module('erl.send').

-export(['erl.send'/2]).

'erl.send'(Pid, Message) ->
  Pid ! Message.
```

Para luego procesar los mensajes que llegan al proceso actual utilizamos la primitiva `receive`, que funciona como la primitiva `case` sobre los mensajes de la casilla de mensajes del proceso. `receive` realiza *pattern-matching* para seleccionar qué estructura debe tener el mensaje que estamos esperando. El proceso luego se quedará esperando hasta recibir un mensaje en la casilla con la estructura dada. Una vez que un mensaje con esa estructura llega al proceso se quita el mensaje de la casilla y se ejecuta la rama del `receive`. Para especificar la estructura del mensaje desde Shen necesitaríamos implementar el *pattern-matching* que realiza Shen directamente sobre Erlang. Para definir esta primitiva (`erl.receive`) no utilizaremos *pattern-matching*, sino que recibiremos el primer mensaje que llegue a la casilla, cualquiera sea su estructura. A su vez, permitimos especificar un tiempo máximo (*timeout*) en milisegundos para esperar por el mensaje, si se excede el tiempo máximo la función retorna el átomo `timeout`.

```
-module('erl.receive').

-export(['erl.receive'/1]).

'erl.receive'(Timeout) ->
  receive
    Message -> Message
  after
    Timeout -> timeout
  end.
```

Finalmente, para poder utilizar las funciones de Erlang directamente sobre Shen introducimos una función global `erl.apply`, que recibe el nombre del módulo, el nombre de la función y una lista de argumentos y aplica la función.

```
-module('erl.apply').

-export(['erl.apply'/3]).

'erl.apply'(Mod, Fun, Args) ->
  erlang:apply(Mod, Fun, Args).
```

Usando estas tres primitivas ahora podemos utilizar toda la capacidad que provee Erlang para escribir programas concurrentes exclusivamente a

través de Shen. El siguiente ejemplo es un programa escrito en Shen que lee los mensajes de la casilla (que se espera sean enteros) hasta recibir un mensaje `stop` y devuelve la suma. Luego este resultado lo mostramos por la salida estándar utilizando la llamada `io:format(''~B'', [Num])`:

```
(define suma_proceso
  Acc ->
    (let (Msg (erl.receive 1000))
         (if (= Msg stop)
             Acc
             (suma_proceso (+ Acc Msg)))))


(erl.apply io format ["~B" [(suma_proceso 0)]])
```

Capítulo 6

Conclusiones

A través de una metodología y formalización de los conceptos y lenguajes utilizados logramos construir un puerto de Shen para Erlang probando en el proceso su corrección.

Por un lado definimos el concepto de "puerto" y todos los elementos involucrados en su construcción, incluyendo los lenguajes y herramientas necesarias. Esta forma de construir nuevas herramientas, utilizando puertos, compiladores e intérpretes que trabajan sobre lenguajes intermedios o "máquinas virtuales", es cada vez más común y aplicable a muchas de las nuevas tecnologías.

Por otro lado, utilizamos un procedimiento que puede ser aplicable para la construcción de cualquier compilador. Mediante este procedimiento no sólo aseguramos su corrección, sino que también sirve como ayuda para la construcción del mismo.

Esta forma de realizar compiladores requiere a su vez una formalización de la semántica operacional de los lenguajes a utilizar ya que, no sólo es necesario para entender cómo funcionan, sino que también lo es para probar la corrección del compilador. Estas formalizaciones de la semántica operacional de los lenguajes utilizados se logra definiendo la semántica del lenguaje de orígen en su totalidad y la semántica del lenguaje de destino de aquellas primitivas que se van a utilizar.

Para este trabajo definimos la semántica de Kλ en su totalidad, y en el caso de Erlang, lo hicimos exclusivamente para aquellas primitivas que fueron utilizadas en la construcción del compilador.

Por último logramos definir de forma informal algunas de las primitivas faltantes del puerto como nuevas primitivas de Shen para que el mismo pueda ser utilizado en ambientes de producción reales, donde se podrán escribir programas en Shen que puedan ser integrados con programas ya existentes en Erlang.

Capítulo 7

Referencias Bibliográficas

[1] P. V. Roy and S. Haridi, *"Concepts, Techniques and Models of Computer Programming"*, 1st ed. Cambridge, Massachusetts, USA: The MIT Press, 2004.

[2] B. W. Kernighan and R. Pike, *The Practice of Programming*, 1st ed. Indianapolis, IN, USA: Addison-Wesley, 1999.

[3] M. Tarver, *"The Book of Shen"*, 3rd ed. Peterborough, Cambridgeshire, UK: Upfront Publishing, 2015.

[4] M. Tarver. "Shen Home Page". [Online]. Available: http://shenlanguage.org/ Accedido: Ago-2018.

[5] M. Tarver, *Logic, Proof and Computation.* Peterborough, Cambridgeshire, UK: Upfront Publishing, 2014.

[6] A. V. Aho, *Compiladores.* Chicago, Illinois, USA: Addison Wesley Longman, 2000.

[7] R. Koeninger. "KLambda - Shen-Language/wiki - GitHub". [Online]. Available: https://github.com/Shen-Language/wiki/wiki/KLambda Accedido: Ago-2018.

[8] B. C. Pierce, *Types and Programming Languages*, 1st ed. Cambridge, Massachusetts, USA: The MIT Press, 2002.

[9] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. Cambridge, Massachusetts, USA: The MIT Press, 1996.

[10] M. Tarver. "GitHub - Shen-Language/shen-cl: Shen for Common Lisp". [Online]. Available: https://github.com/Shen-Language/shen-cl Accedido: Ago-2018.

[11] J. Armstrong, *Programming Erlang: Software for a Concurrent World*, 2nd ed. Raleigh, NC, USA: Pragmatic Bookshelf, 2013.

[12] F. Cesarini and S. Vinoski, *Designing for Scalability with Erlang/OTP: Implement Robust, Fault-Tolerant Systems*, 1st ed. Sebastopol, California, USA: O'Reilly Media, 2016.

Apéndice A

Pruebas de trazabilidad

Recordamos que para probar la corrección y completud de la función *compile* debemos demostrar la siguiente regla para todas las expresiones posibles de Kλ:

$$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e_{kl} \downarrow r_{kl} \rightarrow \langle cconv(\gamma_{kl}),\ fconv(f_{kl}) \rangle \vdash compile(e_{kl},\ \Sigma) \downarrow rconv(r_{kl})$$

que se lee como "para todo e_{kl} que evalúa a un resultado r_{kl} bajo el contexto γ_{kl} y tabla de funciones f, $compile(e_{kl},\ \Sigma)$ debe evaluar al resultado $rconv(r_{kl})$ bajo el contexto $cconv(\gamma_{kl})$ y tabla de funciones $fconv(f)$".

Para probar que esto es cierto para todas las expresiones posibles de Kλ utilizaremos el principio de inducción, es decir, tendremos que probar los casos base (donde no se referencia a compile en el resultado de la función) y los pasos inductivos donde tendremos que probar la corrección partiendo de la base que las expresiones que componen el resultado cumplen con esta regla.

A su vez, para mejorar la legibilidad de estas pruebas, de ahora en adelante utilizaremos los siguientes igualdades:

- $cconv(\gamma_{kl}) \equiv \gamma_{erl}$
- $fconv(f_{kl}) \equiv f_{erl}$
- $rconv(r_{kl}) \equiv r_{erl}$

Las pruebas entonces tendrán la siguiente estructura:

$$\text{Supongamos que } e_{kl} \;=\; \bigcirc. \text{ Debemos probar}$$
$$\langle \gamma_{kl},\ f_{kl} \rangle \vdash \bigcirc \;\downarrow\; r_{kl} \;\rightarrow\; \langle \gamma_{erl},\ f_{erl} \rangle \vdash \mathit{compile}(\bigcirc,\ \gamma) \;\downarrow\; r_{erl}$$
$$\ldots$$

Como mencionamos anteriormente, las pruebas se realizarán de forma informal (narrativa) pero siempre utilizando las estructuras ya definidas (relaciones, funciones y igualdades) para determinar cómo se logra llegar al resultado.

A.1. Pruebas de trazabilidad de expresiones

A.1.1. Corrección de compilación de números, strings y lista vacía

$$\begin{array}{lll} \mathit{compile}(n,\ \Sigma) & = n2n(n) & \text{(Número)} \\ \mathit{compile}(str,\ \Sigma) & = str2str(str) & \text{(String)} \\ \mathit{compile}(\texttt{()},\ \gamma) & = [] & \text{(Lista vacía)} \end{array}$$

Para estos tres casos es suficiente con utilizar la regla que define su semántica en Kλ.

Supongamos que $e_{kl} \;=\; n$. Debemos probar

$$\langle \gamma_{kl},\ f_{kl} \rangle \vdash n \;\downarrow\; r_{kl} \;\rightarrow\; \langle \gamma_{erl},\ f_{erl} \rangle \vdash \mathit{compile}(n,\ \gamma_{kl}) \;\downarrow\; \mathit{rconv}(r_{kl}).$$

Por definición de *compile*, $\mathit{compile}(n) \;=\; n2n(n)$. Por definición de $\downarrow$ la única forma de construir el término de evaluación $\langle \gamma_{kl},\ f_{kl} \rangle \vdash n \;\downarrow\; r_{kl}$ es cuando r_{kl} es $S\langle n \rangle$ (*kl num*), por lo tanto tenemos que probar:

$$\langle \gamma_{erl},\ f_{erl} \rangle \vdash n2n(n) \;\downarrow\; \mathit{rconv}(S\langle n \rangle).$$

Por definición de *rconv* y *vconv*, tenemos que probar:

$$\langle \gamma_{erl},\ f_{erl} \rangle \vdash n2n(n)\ \downarrow\ S\langle n2n(n) \rangle).$$

Esto se prueba utilizando la regla *erl num*, que determina que un número en Erlang evalúa a sí mismo de forma exitosa.

De la misma forma que probamos la corrección para la compilación de los números, también podemos probar la corrección para los strings y la lista vacía, utilizando las reglas *kl string* junto con *erl string* y *kl nil* junto con *erl nil*.

A.1.2. Corrección de compilación de símbolos y variables

$$\begin{array}{lll} compile(s,\ \Sigma) & = s2a(s) & (sis\ \in\ \Sigma)\ \text{(Símbolo a átomo)} \\ compile(s,\ \Sigma) & = s2var(s) & (sis\ \notin\ \Sigma)\ \text{(Símbolo a var)} \end{array}$$

Supongamos que $e_{kl}\ =\ s$, donde s no está contenido en el ambiente Σ.

Debemos probar entonces

$$\langle \gamma_{kl},\ f_{kl} \rangle \vdash s\ \downarrow\ r_{kl}\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile(s,\ \Sigma)\ \downarrow\ rconv(r_{kl}).$$

Por definición de *compile*, $compile(s,\ \Sigma)\ =\ s2a(s)$ cuando s no está presente en Σ. Por definición de $\downarrow$ la única forma de construir el término de evaluación $\langle \gamma_{kl},\ f_{kl} \rangle \vdash s\ \downarrow\ r_{kl}$ es cuando r_{kl} es $S\langle s \rangle$ (*kl sim*), por lo tanto tenemos que probar:

$$\langle \gamma_{erl},\ f_{erl} \rangle \vdash s2a(s)\ \downarrow\ rconv(S\langle s \rangle).$$

Por definición de *rconv* y *vconv*, tenemos que probar:

$$\langle \gamma_{erl},\ f_{erl} \rangle \vdash s2a(s)\ \downarrow\ S\langle s2a(s) \rangle).$$

Esto se prueba utilizando la regla *erl atom*, que determina que un átomo en Erlang evalúa a sí mismo de forma exitosa.

Supongamos ahora que $e_{kl} = s$, donde s sí está contenido en el ambiente Σ.

Debemos probar entonces

$$\langle \gamma_{kl},\ f_{kl} \rangle \vdash s \downarrow r_{kl} \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile(s,\ \Sigma) \downarrow rconv(r_{kl}).$$

Por definición de *compile*, $compile(s,\ \Sigma) = s2var(s)$ cuando s sí está presente en Σ. Por definición de $\downarrow$ la única forma de construir el término de evaluación $\langle \gamma_{kl},\ f_{kl} \rangle \vdash s \downarrow r_{kl}$ es cuando r_{kl} es $S\langle s \rangle$ (*kl sim-var*), por lo tanto tenemos que probar:

$$\langle \gamma_{erl},\ f_{erl} \rangle \vdash s2var(s) \downarrow rconv(S\langle v \rangle).$$

donde v es el valor asociado a s bajo el contexto γ_{kl}. La existencia de este valor está dada por la propiedad que determina que todo elemento en Σ está presente en el contexto γ_{kl}.

Por definición de *rconv* y *vconv*, tenemos que probar:

$$\langle \gamma_{erl},\ f_{erl} \rangle \vdash s2var(s) \downarrow S\langle v \rangle).$$

Teniendo en cuenta que todos los símbolos presentes en γ_{kl} también están presentes como variables en γ_{erl} y asociados a los mismos valores (convertidos con cconv), entonces podemos decir que se cumple: γ_{erl} @! $s2var(s) \longrightarrow v$. Entonces por *erl var* tenemos que la tesis se cumple.

A.1.3. Corrección de compilación de expresiones `if`

$$\begin{array}{l} compile((\texttt{if}\ e\ e_{true}\ e_{false}),\ \Sigma) = \\ \quad \texttt{case}\ compile(e,\ \Sigma)\ \texttt{of} \\ \quad\quad \texttt{true} \rightarrow compile(e_{true}, \Sigma); \\ \quad\quad \texttt{false} \rightarrow compile(e_{false}, \Sigma) \\ \quad \texttt{end} \end{array}$$

Debemos probar entonces

$$\langle \gamma_{kl},\ f_{kl} \rangle \vdash (\texttt{if}\ e\ e_{true}\ e_{false}) \downarrow r_{kl} \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{if}\ e\ e_{true}\ e_{false}),\ \Sigma) \downarrow rconv(r_{kl}).$$

Por definición de $\downarrow$ las tres formas de construir el término de evaluación $\langle \gamma_{kl},\ f_{kl} \rangle \vdash (\texttt{if}\ e\ e_{true}\ e_{false}) \downarrow r_{kl}$ son las siguientes:

- Cuando e evalúa a `true` (*kl if-true*).
- Cuando e evalúa a `false` (*kl if-false*).
- Cuando e evalúa a un error (*kl if-error*).

Vamos a probar cada uno de estos casos:

En el caso que e evalúa exitosamente al valor `true`, por IH entonces $compile(e,\ \Sigma) \downarrow S\langle \texttt{true} \rangle$.

Por *kl if-true* tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e_{true} \downarrow r_{kl-true} \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{if}\ e\ e_{true}\ e_{false}),\ \Sigma) \downarrow rconv(r_{kl-true})$.

En este caso, por la regla *erl case-atom-head* que determina que cuando el átomo de evaluar la expresión de un `case` es igual a la primer rama del case entonces el `case` evalúa al cuerpo de esa rama.

Por lo tanto, tenemos que probar:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e_{true} \downarrow r_{kl-true} \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile(e_{true},\ \Sigma) \downarrow rconv(r_{kl-true})$

que se prueba por IH en e_{true}.

En el caso que e evalúa exitosamente al valor `false`, por IH entonces $compile(e,\ \Sigma) \downarrow S\langle \texttt{false} \rangle$.

Por *kl if-false* tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e_{false} \downarrow r_{kl-false} \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{if}\ e\ e_{true}\ e_{false}),\ \Sigma) \downarrow rconv(r_{kl-false})$.

En este caso, por la regla *erl case-atom-tail* que determina que cuando el átomo de evaluar la expresión de un `case` es distinto a la primer rama del case entonces el `case` evalúa el resto de las ramas para encontrar el átomo correspondiente.

Luego aplicamos la regla *erl case-atom-head* en donde el átomo ahora sí coincide con el átomo de la expresión evaluada (`false`).

Por lo tanto, tenemos que probar:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e_{false} \downarrow r_{kl} \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile(e_{false},\ \Sigma) \downarrow rconv(r_{kl})$

que se prueba por IH en e_{false}.

Para este último caso en el que e evalúa a un error *str*, por IH entonces $compile(e,\ \Sigma) \downarrow E\langle str \rangle$.

Por *kl if-error* tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e \downarrow E\langle str \rangle \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{if}\ e\ e_{true}\ e_{false}),\ \Sigma) \downarrow rconv(E\langle str \rangle)$.

En este caso, por la regla *erl case-atom-error* que determina que cuando la expresión de un `case` evalúa a un error, el mismo es el resultado de evaluación de todo el `case`.

A.1.4. Corrección de compilación de expresiones `cond`

$$
\begin{array}{l}
compile((\texttt{cond}\ (e\ e_{body}) \ldots rest),\ \Sigma) = \\
\quad \texttt{case}\ compile(e,\ \Sigma)\ \texttt{of} \\
\quad\quad \texttt{true} \rightarrow compile(e_{body}, \Sigma); \\
\quad\quad \texttt{false} \rightarrow compile((\texttt{cond} \ldots rest), \Sigma) \\
\quad \texttt{end}
\end{array}
$$

Debemos probar entonces

$$\langle \gamma_{kl},\ f_{kl} \rangle \vdash (\texttt{cond}\ (e\ e_{body})\ \ldots rest) \downarrow r_{kl} \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash$$

$$compile((\texttt{cond}\ (e\ e_{body})\ \ldots rest),\ \Sigma)\ \downarrow\ rconv(r_{kl}).$$

Por definición de $\downarrow$ las tres formas de construir el término de evaluación $\langle \gamma_{kl},\ f_{kl} \rangle \vdash (\texttt{cond}\ (e\ e_{body})\ \ldots rest)\ \downarrow\ r_{kl}$ son las siguientes:

- Cuando e evalúa a `true` (*kl cond-true*).
- Cuando e evalúa a `false` (*kl cond-false*).
- Cuando e evalúa a un error (*kl cond-error*).

Vamos a probar cada uno de estos casos:

En el caso que e evalúa exitosamente al valor `true`, por IH entonces $compile(e,\ \Sigma)\ \downarrow\ S\langle \texttt{true} \rangle$.

Por *kl cond-true* tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e_{true}\ \downarrow\ r_{kl-true}\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{cond}\ (e\ e_{body})\ \ldots rest),\ \Sigma)\ \downarrow\ rconv(r_{kl})$.

En este caso, por la regla *erl case-atom-head* que determina que cuando el átomo de evaluar la expresión de un `case` es igual a la primer rama del case entonces el `case` evalúa al cuerpo de esa rama.

Por lo tanto, tenemos que probar:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e_{body}\ \downarrow\ r_{kl-body}\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile(e_{body},\ \Sigma)\ \downarrow\ rconv(r_{kl-body})$

que se prueba por IH en e_{body}.

En el caso que e evalúa exitosamente al valor `false`, por IH entonces $compile(e,\ \Sigma)\ \downarrow\ S\langle \texttt{false} \rangle$.

Por *kl cond-false*) tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash (\texttt{cond}\ \ldots rest)\ \downarrow\ r_{kl-rest}\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{cond}\ (e\ e_{body}\ \ldots rest),\ \Sigma)\ \downarrow\ rconv(r_{kl-rest})$.

En este caso, aplicando la regla *erl case-atom-tail* y luego la regla *erl case-atom-head* tenemos que probar entonces que la segunda rama (del resto de las expresiones de cond) evalúa a $rconv(r_{kl-rest})$.

Por lo tanto, tenemos que probar:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash (\texttt{cond} \ldots rest) \downarrow r_{kl-rest} \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{cond} \ldots rest),\ \Sigma) \downarrow rconv(r_{kl-rest})$

que se prueba por IH en $(\texttt{cond} \ldots rest)$.

Para este último caso en el que e evalúa a un error *str*, por IH entonces $compile(e,\ \Sigma) \downarrow E\langle str \rangle$.

Por *kl cond-error* tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e \downarrow E\langle str \rangle \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{cond}\ (e\ e_{body})\ \ldots rest),\ \Sigma) \downarrow rconv(E\langle str \rangle)$.

En este caso, por la regla *erl case-atom-error* que determina que cuando la expresión de un `case` evalúa a un error, el mismo es el resultado de evaluación de todo el `case`, como se quiere demostrar.

A.1.5. Corrección de compilación de expresiones `and` y `or`

Corrección de compilación de expresiones `and`

$$
\begin{array}{l}
compile((\texttt{and}\ e_1\ e_2),\ \Sigma) = \\
\quad \texttt{case}\ compile(e_1,\ \Sigma)\ \texttt{of} \\
\quad\quad \texttt{true} \rightarrow compile(e_2, \Sigma); \\
\quad\quad \texttt{false} \rightarrow \texttt{false} \\
\quad \texttt{end}
\end{array}
$$

Por definición de $\downarrow$ las tres formas de construir el término de evaluación $\langle \gamma_{kl},\ f_{kl} \rangle \vdash (\texttt{and}\ e_1\ e_2) \downarrow r_{kl}$ son las siguientes:

- Cuando e_1 evalúa a `true` (*kl and-true*).
- Cuando e_1 evalúa a `false` (*kl and-false*).
- Cuando e_1 evalúa a un error (*kl and-error*).

Vamos a probar cada uno de estos casos:

En el caso que e_1 evalúa exitosamente al valor `true`, por IH entonces $compile(e_1,\ \Sigma)\ \downarrow\ S\langle \texttt{true} \rangle$.

Por *kl and-true* tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e_2\ \downarrow\ r_{kl-e2}\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{and}\ e_1\ e_2),\ \Sigma)\ \downarrow\ rconv(r_{kl-e2})$.

En este caso, por la regla *erl case-atom-head* tenemos que probar:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash compile(e_2,\ \Sigma)\ \downarrow\ r_{kl-e2}\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile(e_2,\ \Sigma)\ \downarrow\ rconv(r_{kl-e2})$.

que se prueba por IH en e_2.

En el caso que e evalúa exitosamente al valor `false`, por IH entonces $compile(e_1,\ \Sigma)\ \downarrow\ S\langle \texttt{false} \rangle$.

Por *kl and-false* tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash \texttt{false}\ \downarrow\ r_{kl-false}\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{and}\ e_1\ e_2),\ \Sigma)\ \downarrow\ rconv(r_{kl-false})$.

En este caso, aplicando la regla *erl case-atom-tail* y luego aplicando la regla *erl case-atom-head* tenemos que probar:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash \texttt{false}\ \downarrow\ r_{kl-false}\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash \texttt{false}\ \downarrow\ rconv(r_{kl-false})$.

que se prueba por IH en `false`.

Para este último caso en el que e evalúa a un error *str*, por IH entonces $compile(e_1,\ \Sigma)\ \downarrow\ E\langle str \rangle$.

Por *kl and-error*) tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e_1\ \downarrow\ E\langle str \rangle\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{and}\ e_1\ e_2),\ \Sigma)\ \downarrow\ rconv(E\langle str \rangle)$.

En este caso, por la regla *erl case-atom-error* que determina que cuando la expresión de un `case` evalúa a un error, el mismo es el resultado de evaluación

de todo el `case`.

Corrección de compilación de expresiones `or`

$$
\begin{array}{l}
\mathit{compile}((\texttt{or}\ e_1\ e_2),\ \Sigma) = \\
\quad \texttt{case}\ \mathit{compile}(e_1,\ \Sigma)\ \texttt{of} \\
\quad\quad \texttt{false}\ \rightarrow\ \mathit{compile}(e_2,\ \Sigma); \\
\quad\quad \texttt{true}\ \rightarrow\ \texttt{true} \\
\quad \texttt{end}
\end{array}
$$

Por definición de $\downarrow$ las tres formas de construir el término de evaluación $\langle \gamma_{kl},\ f_{kl} \rangle \vdash (\texttt{and}\ e_1\ e_2)\ \downarrow\ r_{kl}$ son las siguientes:

- Cuando e_1 evalúa a `true` (*kl or-true*).
- Cuando e_1 evalúa a `false` (*kl or-false*).
- Cuando e_1 evalúa a un error (*kl or-error*).

Vamos a probar cada uno de estos casos:

En el caso que e_1 evalúa exitosamente al valor `true`, por IH entonces $\mathit{compile}(e_1,\ \Sigma)\ \downarrow\ S\langle\texttt{true}\rangle$.

Por *kl or-true* tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash \texttt{true}\ \downarrow\ r_{kl-e2}\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash \mathit{compile}((\texttt{and}\ e_1\ e_2),\ \Sigma)\ \downarrow\ \mathit{rconv}(r_{kl-e2})$.

En este caso, por la regla *erl case-atom-head* tenemos que probar:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash \texttt{true}\ \downarrow\ r_{kl-true}\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash \texttt{true}\ \downarrow\ \mathit{rconv}(r_{kl-true})$.

que se prueba por IH en `false`.

En el caso que e evalúa exitosamente al valor `false`, por IH entonces $\mathit{compile}(e_1,\ \Sigma)\ \downarrow\ S\langle\texttt{false}\rangle$.

Por *kl and-false* tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e_2 \downarrow r_{kl-false} \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{and}\ e_1\ e_2),\ \Sigma) \downarrow rconv(r_{kl-false})$.

En este caso, aplicando la regla *erl case-atom-tail* y luego aplicando la regla *erl case-atom-head* tenemos que probar:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e_2 \downarrow r_{kl-e2} \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash (compile\ e_2,\ \Sigma) \downarrow rconv(r_{kl-e2})$.

que se prueba por IH en e_2.

Para este último caso en el que e evalúa a un error *str*, por IH entonces $compile(e_1,\ \Sigma) \downarrow E\langle str \rangle$.

Por *kl and-error* tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e_1 \downarrow E\langle str \rangle \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{and}\ e_1\ e_2),\ \Sigma) \downarrow rconv(E\langle str \rangle)$.

En este caso, por la regla *erl case-atom-error* que determina que cuando la expresión de un `case` evalúa a un error, el mismo es el resultado de evaluación de todo el `case`.

A.1.6. Corrección de compilación de expresiones `lambda`

$$compile((\texttt{lambda}\ s_{param}\ e_{body}),\ \Sigma) = \\ \texttt{fun}\ (s2var(s_{param}))\ \rightarrow\ compile(e_{body},\ \Sigma\ \leftarrow\ s)\ \texttt{end}$$

En este caso, la variable de la función se agrega al ambiente Σ para que dentro del cuerpo de la función el símbolo s se interprete como variable.

Tenemos que probar:

$$\langle \gamma_{kl},\ f_{kl} \rangle \vdash (\texttt{lambda}\ s_{param}\ e_{body}),\ \Sigma) \downarrow r_{kl} \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash \\ compile((\texttt{lambda}\ s_{param}\ e_{body}),\ \Sigma),\ \gamma_{kl}) \downarrow rconv(r_{kl}).$$

Esto se prueba por definición *kl lambda* y luego de *vconv* y *rconv*.

A.1.7. Corrección de compilación de expresiones `let`

$$
\begin{array}{l}
compile((\texttt{let}\ s_{var}\ e\ e_{body}),\ \Sigma) = \\
\quad \texttt{case}\ compile(e,\ \Sigma)\ \texttt{of} \\
\quad\quad s2var(s)\ \rightarrow\ compile(e_{body},\ \Sigma\ \longleftarrow\ s) \\
\quad \texttt{end}
\end{array}
$$

Por definición de $\downarrow$ las tres formas de construir el término de evaluación $\langle \gamma_{kl},\ f_{kl} \rangle \vdash (\texttt{and}\ e_1\ e_2)\ \downarrow\ r_{kl}$ son las siguientes:

- Cuando e evalúa a un valor exitoso `v` (*kl let*).
- Cuando e evalúa a un error (*kl let-error*).

Vamos a probar cada uno de estos casos:

En el caso que e evalúa exitosamente a un valor v, entonces:

- Por aplicar *kl let* tenemos que $\gamma_{kl}\ \longleftarrow\ (s_{var},\ v)\ \equiv\ \gamma'_{kl}$ y $\langle \gamma'_{kl},\ f \rangle \vdash e_{body}\ \downarrow\ r_{kl-body}$.
- Por IH tenemos que $\langle \gamma_{kl},\ f_{kl} \rangle e\ \downarrow\ \langle v \rangle \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle compile(e,\ \Sigma)\ \downarrow\ S\langle vconv(v) \rangle$.

Por *kl let* tenemos que probar entonces:

$\langle \gamma'_{kl},\ f_{kl} \rangle \vdash e_{body}\ \downarrow\ r_{kl-body}\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{let}\ s_{var} e\ e_{body}),\ \Sigma)\ \downarrow\ rconv(r_{kl-body})$.

Por *erl case-var*, que dice que evaluar un `case` del tipo variable con una única rama es evaluar esa rama con la variable agregada al contexto, tenemos que probar entonces:

$\langle \gamma'_{kl},\ f_{kl} \rangle \vdash e_{body}\ \downarrow\ r_{kl-body}\ \rightarrow\ \langle \gamma'_{erl},\ f_{erl} \rangle \vdash compile(e_{body},\ \Sigma\ \longleftarrow\ s_{var})\ \downarrow\ rconv(r_{kl-body})$

donde $\gamma'_{erl} \equiv \gamma_{erl}\ \longleftarrow\ (s2var(s_{var}),\ vconv(v))$.

Por definición de cconv $\gamma'_{erl} \equiv cconv(\gamma_{kl}\ \longleftarrow\ (s_{var},\ v))$, entonces por IH tenemos que $\langle \gamma'_{kl},\ f_{kl} \rangle \vdash e_{body}\ \downarrow\ \langle v \rangle\ \rightarrow\ \langle \gamma'_{erl},\ f_{erl} \rangle \vdash compile(e,\ \Sigma)\ \downarrow\ S\langle vconv(v) \rangle$.

En el otro caso, en donde e evalúa a un error *str*, por IH entonces $compile(e,\ \Sigma)\ \downarrow\ E\langle str\rangle$.

Por *kl let-error* tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl}\rangle \vdash e\ \downarrow\ E\langle str\rangle\ \rightarrow\ \langle \gamma_{erl},\ f_{erl}\rangle \vdash compile((\texttt{let}\ s_{var}\ e\ e_{body}),\ \Sigma)\ \downarrow\ rconv(E\langle str\rangle)$.

En este caso, por la regla *erl case-var-error* que determina que cuando la expresión de un `case` del tipo variable evalúa a un error, el mismo es el resultado de evaluación de todo el `case`, como se quiere demostrar.

A.1.8. Corrección de compilación de "valores congelados"

Estas expresiones compilan a una función anónima de Erlang sin parámetros para luego ser llamada y evaluada.

$$compile((\texttt{freeze}\ e_{body}),\ \Sigma) = \\ \texttt{fun}\ ()\ \rightarrow\ compile(e_{body},\ \Sigma)\ \texttt{end}$$

Tenemos que probar:

$$\langle \gamma_{kl},\ f_{kl}\rangle \vdash (\texttt{freeze}\ e_{body}),\ \Sigma)\ \downarrow\ r_{kl}\ \rightarrow\ \langle \gamma_{erl},\ f_{erl}\rangle \vdash \\ compile((\texttt{freeze}\ e_{body}),\ \Sigma),\ \gamma_{kl})\ \downarrow\ rconv(r_{kl}).$$

Esto se prueba por definición *kl freeze* y luego de *vconv* y *rconv*.

A.1.9. Corrección de compilación de `trap-error`

Estas expresiones compilan a una estructura `try` de Erlang con una rama para el caso exitoso y una rama para el caso de error.

$$
\begin{array}{l}
compile((\texttt{trap-error}\ e\ e_{handler}),\ \Sigma) = \\
\quad \texttt{try}\ compile(e,\ \Sigma)\ \texttt{of} \\
\qquad s2var(sv)\ \rightarrow\ s2var(sv) \\
\quad \texttt{catch} \\
\qquad s2var(serr)\ \rightarrow\ compile((e_{handler}\ serr),\ \Sigma \longleftarrow serr) \\
\quad \texttt{end}
\end{array}
$$

Debemos probar entonces

$$
\begin{array}{c}
\langle \gamma_{kl},\ f_{kl} \rangle \vdash (\texttt{trap-error}\ e\ e_{handler}),\ \Sigma)\ \downarrow\ r_{kl}\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash \\
compile((\texttt{trap-error}\ e\ e_{handler}),\ \Sigma),\ \gamma_{kl})\ \downarrow\ rconv(r_{kl}).
\end{array}
$$

Por definición de $\downarrow$ las dos formas de construir el término de evaluación $\langle \gamma_{kl},\ f_{kl} \rangle \vdash (\texttt{trap-error}\ e\ e_{handler})\ \downarrow\ r_{kl}$ son las siguientes:

- Cuando e evalúa a un valor exitoso v (*kl trap-error-success*).
- Cuando e evalúa a un error str (*kl trap-error-failure*).

Vamos a probar cada uno de estos casos:

En el caso que e evalúa exitosamente a un valor v, por IH entonces $compile(e,\ \Sigma)\ \downarrow\ S\langle v \rangle$.

Por *kl trap-error-success* tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e\ \downarrow\ S\langle v \rangle\ \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{trap-error}\ e\ e_{handler}),\ \Sigma)\ \downarrow\ rconv(S\langle v \rangle)$.

Luego aplicamos la regla *erl try-success*. Tenemos que probar ahora:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash e\ \downarrow\ S\langle v \rangle\ \rightarrow\ \langle \gamma_{erl} \longleftarrow (var,\ v),\ f_{erl} \rangle \vdash var\ \downarrow\ rconv(S\langle v \rangle)$

donde *var* es una nueva variable única en el programa. Por definición de *erl var* y *lookup ctx head*, entonces obtenemos que *var* evalúa a $S\langle v \rangle$.

En el caso que e evalúa a un error *str*, por IH entonces $compile(e,\ \Sigma)\ \downarrow\ E\langle str \rangle$.

Por *kl trap-error-failure* tenemos que probar entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash (e_{handler}\ str) \downarrow r_{kl-handler} \rightarrow \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((\texttt{trap-error}\ e\ e_{handler}),\ \Sigma) \downarrow rconv(r_{kl-handler})$.

Luego aplicamos la regla *erl try-failure*. Tenemos que probar ahora:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash (e\ str) \downarrow r_{kl-handler} \rightarrow \langle \gamma_{erl} \longleftarrow (var_{err},\ str),\ f_{erl} \rangle \vdash compile((e_{handler}\ var_{err}),\ \Sigma \longleftarrow var_{err}) \downarrow rconv(r_{kl-handler})$

donde var_{err} es una nueva variable única en el programa. Nuevamente, por definición de *erl var* y *lookup ctx head*, entonces obtenemos que var_{err} evalúa a $S\langle str \rangle$.

Por IH entonces podemos decir que $\langle \gamma_{erl} \longleftarrow (var_{err},\ str),\ f_{erl} \rangle \vdash compile((e_{handler}\ var_{err}),\ \Sigma \leftarrow var_{err}) \downarrow rconv(r_{kl-handler})$.

A.1.10. Corrección de compilación de aplicación de funciones

Corrección de compilación de aplicación de "valores congelados"

$$compile((e),\ \Sigma) = compile(e,\ \Sigma)()$$

Por definición de $\downarrow$ las dos formas de construir el término de evaluación $\langle \gamma_{kl},\ f_{kl} \rangle \vdash (e) \downarrow r_{kl}$ son las siguientes:

- Cuando e evalúa a un valor exitoso "congelado" $\{(\texttt{freeze}\ e_{body}),\ \gamma'_{kl}\}$ (*kl app-freeze*).
- Cuando e evalúa a un error (*kl app-freeze-error*).

Vamos a probar cada uno de estos casos:

En el caso que e evalúa al valor $\{(\texttt{freeze}\ e_{body}),\ \gamma'_{kl}\}$, por IH entonces $compile(e,\ \Sigma) \downarrow S\langle \texttt{fun}\ () \rightarrow compile(e_{body},\ \Sigma)\ \texttt{end},\ conv(\gamma'_{kl}) \rangle$

Por *kl app-freeze* tenemos que probar entonces:

$\langle \gamma'_{kl},\ f_{kl} \rangle \vdash e_{body} \downarrow r_{kl} \rightarrow \langle cconv(\gamma'_{kl}),\ f_{erl} \rangle \vdash (\texttt{fun}\ () \rightarrow compile(e_{body},\ \Sigma)\ \texttt{end})() \downarrow rconv(r_{kl})$.

Por *erl app-anon* tenemos que probar:

$\langle \gamma'_{kl},\ f_{kl} \rangle \vdash e_{body} \ \downarrow \ r_{kl} \ \rightarrow \ \langle cconv(\gamma'_{kl}),\ f_{erl} \rangle \vdash compile(e_{body},\ \Sigma) \ \downarrow rconv(r_{kl})$

que surge por IH.

Para el caso en el que e evalúa a un error *str*, por IH entonces $compile(e,\ \Sigma) \downarrow E\langle str \rangle$.

Por *kl app-freeze-error* entonces:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash (e) \ \downarrow \ E\langle str \rangle \ \rightarrow \ \langle \gamma_{kl},\ f_{erl} \rangle \vdash compile((e),\ \Sigma) \downarrow rconv(r_{kl})$.

En este caso, por la regla *erl app-error-efun* que determina que cuando en una aplicación la evaluación del operando resulta en un error entonces toda la aplicación evalúa a ese error.

Corrección de compilación de aplicación de funciones lambda

$$compile((e_1\ e_2),\ \Sigma) = compile(e_1,\ \Sigma)(compile(e_2,\ \Sigma))$$
$$compile((e_1\ e_2\ \overline{e_{rest}}),\ \Sigma) = compile(((e_1\ e_2)\ \overline{e_{rest}}),\ \Sigma)$$

En este caso sólo es necesario probar la primer forma de la expresión $((e_1 e_2))$, ya que la segunda forma $((e_1\ e_2\ \overline{e_{rest}}))$ se da por *kl app-left-assoc*.

Por definición de $\downarrow$ las tres formas de construir el término de evaluación $\langle \gamma_{kl},\ f_{kl} \rangle \vdash (e_1\ e_2) \ \downarrow \ r_{kl}$ son las siguientes:

- Cuando e_1 evalúa a un término `lambda` y e_2 evalúa a un término exitoso (*kl app-lambda*).
- Cuando e_1 evalúa a un error (*kl app-lambda-e1-error*).
- Cuando e_1 evalúa a un término `lambda` y e_2 evalúa a un error (*kl app-lambda-e2-error*).

Vamos a probar cada uno de estos casos:

En el caso que e_1 evalúa exitosamente a una expresión lambda $\langle (\texttt{lambda}\ s_{param}\ e_{body}),\ \gamma'_{kl} \rangle$ y e_2 evalúa exitosamente a un valor v, por IH entonces $compile(e_1,\ \Sigma) \ \downarrow$

$S\langle\langle$`fun`$(s2var(s_{param})) \rightarrow compile(e_{body},\ vars(\gamma'_{kl}))$ `end`$,\ cconv(\gamma'_{kl})\rangle$ y $compile(e_2,\ \Sigma)\ \downarrow S\langle vconv(v)\rangle$.

Para este caso podemos aplicar entonces la regla *erl app-anon* que establece que dado un nuevo contexto que surge de agregar la asignación de la variable $s2var(s_{param})$ y el valor *vconv(v)* al contexto $cconv(\gamma')$. Tenemos que probar entonces:

$\langle\gamma'_{kl} \longleftarrow (s_{param}, v),\ f_{kl}\rangle \vdash e_{body} \downarrow r_{kl} \rightarrow \langle\gamma' \longleftarrow (s2var(s_{param}), vconv(v)),\ f_{erl}\rangle \vdash compile(e_{body},\ vars(\gamma'_{kl}) \longleftarrow s_{param}) \downarrow r_{erl}$

que se prueba por IH en e_{body}.

En el caso que e_1 evalúa a un error *str*, por IH entonces $compile(e_1,\ \Sigma)\ \downarrow E\langle str\rangle$.

Por *kl app-lambda-e1-error* tenemos que probar entonces:

$compile((e_1,\ e_2),\ \Sigma)\ \downarrow\ E\langle str\rangle$.

Este caso se prueba por la regla *erl app-error-efun*, que determina que cuando el operando evalúa a un error toda la aplicación evalúa a ese error.

En el caso que e_1 evalúa a un valor exitoso, pero e_2 evalúa a un error *str*, por IH entonces $compile(e_1,\ \Sigma)\ \downarrow\ S\langle\langle$`fun` $(var_1) \rightarrow e_{body}$ `end`$,\ \gamma'\rangle\rangle$ y $compile(e_2,\ \Sigma)\ \downarrow\ E\langle str\rangle$.

Por *kl app-lambda-e2-error* tenemos que probar entonces:

$compile((e_1,\ e_2),\ \Sigma)\ \downarrow\ E\langle str\rangle$.

Este caso se prueba por la regla *erl app-error-params*, que determina que cuando el uno de los parámetros evalúa a un error toda la aplicación evalúa a ese error.

Corrección de compilación de aplicación de funciones globales

En este caso la compilación depende del número de parámetros que recibe la función global y el número de parámetros que se tienen al aplicar esta función.

Los tres casos de compilación vistos anteriormente, para una función global s que recibe m parámetros son los siguientes:

$$
\begin{array}{ll}
compile((s\ e_1\ \ldots\ e_n),\ \Sigma) = & \text{si } n \leq m \\
\quad \texttt{fun}\ (s2var(sv)) \rightarrow compile((s\ e_1\ \ldots\ e_n\ sv),\ \Sigma \longleftarrow sv)\ \texttt{end} & \\
compile((s\ e_1\ \ldots\ e_n),\ \Sigma) = & \text{si } n = m \\
\quad s2funname(s)(compile(e_1,\ \Sigma),\ \ldots,\ compile(e_n,\ \Sigma))) & \\
compile((s\ e_1\ \ldots\ e_n),\ \Sigma) = & \text{si } n \geq m \\
\quad compile(((s\ take(m,\ e_1,\ \ldots\ e_n))\ skip(m,\ e_1,\ \ldots\ e_n))) &
\end{array}
$$

donde *sv* es una variable nueva que se obtiene a través de la función *newvar* ($sv = newvar(\Sigma)$) y *take* y *skip* son las funciones auxiliares definidas anteriormente.

Recordemos también que las funciones en la tabla de funciones al evaluar las expresiones Erlang contiene las funciones de Kλ "de-currificadas" ($decurrify(compile(e_{kl},\ \epsilon))$).

Para los tres casos, la única forma de evaluar estos términos en Kλ es utilizando las reglas *kl app-left-assoc* y *kl eval-global-fun*, donde $e_{fun} \equiv f_{kl}$ @! s:

$$\langle \gamma_{kl},\ f_{kl} \rangle \vdash (s\ e_1\ \ldots\ e_n)\ \downarrow\ r_{kl} \rightarrow \langle \gamma_{kl},\ f_{kl} \rangle \vdash ((e_{fun}\ e_1)\ \ldots\ e_n)\ \downarrow\ r_{kl}$$

Para el primer caso, donde la cantidad de parámetros que se aplican es menor a la aridad de la función tenemos que probar:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash ((e_{fun}\ e_1)\ \ldots\ e_n)\ \downarrow\ r_{kl} \rightarrow\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash \texttt{fun}\ (s2var(sv)) \rightarrow compile((s\ e_1\ \ldots\ e_n\ sv),\ \Sigma \longleftarrow sv)\ \texttt{end} \downarrow\ rconv(r_{kl})$.

Como esta función aumenta, no tenemos forma de utilizar la IH, por lo que tendremos que recurrir a encontrar una equivalencia entre la expresión en Kλ y su expresión compilada. En este caso, como sabemos que $n \leq m$, entonces el resultante de $((e_{fun}\ e_1)\ \ldots\ e_n)$ es una expresión de la forma ($\texttt{lambda}\ s_{n+1}\ e_{fun_{n+1}}$). Para este caso, en la compilación de este término, s_{n+1} sería *s2var(sv)* y $e_{fun_{n+1}}$ sería $compile(s\ e_1\ \ldots\ e_n\ sv,\ \Sigma\ \longleftarrow sv)$.

Para el segundo caso, donde la cantidad de parámetros que se aplican es igual a la aridad de la función tenemos que probar:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash ((e_{fun}\ e_1)\ \ldots\ e_n)\ \downarrow\ r_{kl} \rightarrow$
$\langle \gamma_{erl},\ f_{erl} \rangle \vdash s2funname(s)(compile(e_1,\ \Sigma),\ \ldots,\ compile(e_n,\ \Sigma)) \downarrow rconv(r_{kl})$.

Para este caso, como aplicar la función currificada con todos sus parámetros es igual a aplicar a la función no-currificada, entonces evaluar esta expresión es evaluar el cuerpo de la misma:

$\langle \gamma_{kl},\ f_{kl} \rangle \vdash ((e_{fun-kl}\ e_1)\ \ldots\ e_n)\ \downarrow\ r_{kl} \rightarrow$

$$\langle \gamma_{erl},\ f_{erl} \rangle \vdash e_{fun-erl}(compile(e_1,\ \Sigma),\ \ldots,\ compile(e_n,\ \Sigma)) \downarrow rconv(r_{kl}).$$

Por IH el resultado de compilar cada uno de los términos $e_1\ \ldots\ e_n$ resulta en los mismos valores exitosos (o errores), por lo que el resultado de realizar la aplicación de los mismos parámetros sobre una función anónima equivalente resultará en lo mismo.

Para el tercer caso, donde la cantidad de parámetros que se aplican es mayor a la aridad de la función tenemos que probar:

$$\langle \gamma_{kl},\ f_{kl} \rangle \vdash ((e_{fun}\ e_1)\ \ldots\ e_m\ \ldots\ e_n)\ \downarrow\ r_{kl} \rightarrow \\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile(((s\ e_1,\ \ldots\ e_m)\ e_{m+1},\ \ldots\ e_n),\ \Sigma).$$

El primer término $((e_{fun}\ e_1)\ \ldots\ e_m)$ resulta ser el caso anterior, donde el número de parámetros es igual a la aridad de la función, por lo tanto, hay que probar

$$\langle \gamma_{kl},\ f_{kl} \rangle \vdash (e_{fun-res-kl}\ e_{m+1}\ \ldots\ e_n)\ \downarrow\ r_{kl} \rightarrow \\ \langle \gamma_{erl},\ f_{erl} \rangle \vdash compile((e_{fun-res-kl}\ e_{m+1}\ \ldots\ e_n),\ \Sigma)$$

que se prueba utilizando la misma prueba de aplicaciones de funciones `lambda`.

Apéndice B

Funciones y símbolos globales de Kλ

De las 57 primitivas de Kλ, 48 de ellas son funciones y símbolos globales que uno debe definir para que el puerto funcione correctamente. A continuación proveemos la lista, el funcionamiento y una breve descripción de cómo fueron implementadas.

B.1. Funciones

A continuación se describe la forma en que fueron implementadas las 48 funciones y símbolos globales necesarios completar el puerto de Shen. Estas funciones y símbolos globales, junto con las "formas especiales" de Kλ constituyen las 57 primitivas a implementar para completar el puerto en su totalidad.

Para entender los posibles valores que reciben y retornan estas primitivas introducimos el concepto de tipos en Kλ de forma informal. A su vez, agregamos nuevos posibles valores, que si bien no son útiles para definir la semántica de Kλ, se necesitan para definir estas primitivas.

Un tipo lo definimos como un conjunto de valores posibles. Los tipos que vamos a utilizar son:

- `number`, `symbol`, `string` - Los números, símbolos y cadenas de texto que vimos como posible valor de Kλ.

- `boolean` - Los símbolos `true` y `false`. Estos símbolos son tanto del tipo `symbol` como del tipo `boolean`.
- `cons` - Pares de valores.
- `vector` - Vectores de largo fijo (n), cuyos valores son accedidos y modificados a través de un índice i ($0 \leq i < n$).
- `stream` - *Streams* (o archivos) del sistema operativo.
- `exception` - Excepciones que al ser lanzadas, pueden luego ser capturadas por la primitiva `trap-error`.
- `any` - Cualquier posible valor de Kλ.

A continuación detallamos estas primitivas agrupadas por la funcionalidad que proveen.

B.1.1. Funciones sobre booleanos

and : `boolean` $\rightarrow$ `boolean` $\rightarrow$ `boolean` $\rightarrow$ `any`

or : `boolean` $\rightarrow$ `boolean` $\rightarrow$ `boolean` $\rightarrow$ `any`

if : `boolean` $\rightarrow$ `any` $\rightarrow$ `any` $\rightarrow$ `any`

Estas tres funciones actúan como las formas especiales `and` y `or`, pero actúan como funciones currificadas normales, sin el "corto-circuitado". Deben recibir todos los parámetros para ser evaluadas. Su implementación se logra de forma directa con Erlang utilizando la expresión `case`.

B.1.2. Funciones para manejo de errores

$simple-error$: `string` $\rightarrow$ `bottom`

Para lanzar excepciones en Kλ se deberá utilizar esta función `simple-error` que recibe un `string` describiendo la excepción. Esta excepción deberá ser capturada por una expresión `trap-error`. En Erlang, esto se logra con la llamada a la función `throw`.

$error-to-string$: `exception` $\rightarrow$ `string`

Las excepciones de Kλ las representamos en Erlang como cadenas de texto, por lo que esta función la definimos como la identidad.

B.1.3. Funciones para manejo de símbolos globales

Los símbolos globales son una estructura de datos tabular mutable, donde cada las claves son símbolos y los valores son cualquier valor posible de Kλ. Para implementarlo utilizamos la librería *ets* que Erlang provee, que permite acceder y modificar valores Erlang de forma tabular donde las claves son átomos y los valores son cualquier valor posible de Erlang.

Al final de este apéndice se detallan los símbolos globales que deben estar pre-definidos para ejecutar programas Shen correctamente.

Las funciones que se necesitan para el manejo de símbolos globales son:

set : `symbol` → `any` → `any`

Asigna un símbolo global al valor dado y retorna este valor.

value : `symbol` → `any`

Obtiene el valor asociado al símbolo dado. Lanza un error si el mismo no se encuentra en la tabla global.

B.1.4. Funciones para manejo de números

number? : `any` → `boolean`

Retorna `true` si el parámetro es un número, `false` en cualquier otro caso.

`+`, `-`, `*`, `/` : `number` → `number` → `number`

Las funciones aritméticas sobre números.

>, <, >=, <= : number → number → boolean

Las funciones básicas de comparación de números. Estas operaciones se implementan utilizando los operadores infijos de expresiones.

B.1.5. Funciones para manejo de cadenas de caracteres

Las cadenas de caracteres las implementamos en el puerto como la tupla $string, < s >$, siendo $<s>$ un string de Erlang. Todas las operaciones sobre estas cadenas son fácilmente implementadas con *pattern-matching*.

string? : `any` → `boolean`

Retorna `true` si el parámetro es un número, `false` en cualquier otro caso.

pos : `string` → `number` → `string`

Retorna el carácter en la posición dada como una cadena de texto largo 1. Lanza un error si el índice es mayor al largo de la cadena.

tlstr : `string` → `string`

Retorna un nuevo string con todos los caracteres del string dado, sin el primer carácter. Lanza un error si la cadena de caracteres dada es vacía.

cn : `string` → `string` → `string`

Retorna el resultado de concatenar los dos strings dados.

str : `any` → `string`

Retorna la representación como cadena de caracteres de cualquier valor posible de Kλ.

$string- > n$: `string` → `number`

Retorna el primer carácter de la cadena dada en su representación numérica. Esta representación numérica dependerá de la codificación, por lo que se deja liberado al puerto. Lanza un error si la cadena de caracteres dada es vacía.

$n- > string$: `number` → `string`

Retorna la cadena de caracteres con el carácter dado en representación numérica.

B.1.6. Funciones para manejo de vectores

Los vectores en Kλ son de largo fijo y se representan como una estructura mutable en donde se modifican y obtienen sus valores a través de un índice.

En Kλ estos vectores se llaman *absvectors*.

Para representar estos valores utilizamos procesos de Erlang, donde cada vector es un nuevo proceso que mantiene el estado con los valores del vector y recibe mensajes de forma asíncrona para acceder y modificar estos valores. Esta forma de simular la mutabilidad a través de procesos se da por la semántica que define el comportamiento de los procesos y los mensajes que se envían entre ellos [1].

absvector : `number` $\rightarrow$ `vector`

Construye el vector con el largo dado.

address$->$: `vector` $\rightarrow$ `number` $\rightarrow$ `any` $\rightarrow$ `vector`

Asigna el valor dado en el índice dado. Lanza un error si el índice es mayor o igual al largo del vector.

$<-$*address* : `vector` $\rightarrow$ `number` $\rightarrow$ `any`

Obtiene el valor del vector en el índice dado. Lanza un error si el índice es mayor o igual al largo del vector.

absvector? : `any` $\rightarrow$ `boolean`

Retorna `true` si el parámetro es un vector, `false` en cualquier otro caso.

B.1.7. Funciones para manejo de pares

Los pares en Kλ los representamos como tuplas en Erlang. Por ejemplo, el par `(cons 1 2)` lo representamos como `{cons, 1, 2}`. Para formar listas se utiliza esta función `cons`, por lo que las listas en Kλ son representadas como pares anidados en Erlang, no se utilizan listas de Erlang.

cons? : `any` $\rightarrow$ `boolean`

Retorna `true` si el parámetro es un par, `false` en cualquier otro caso.

cons : `any` $\rightarrow$ `any` $\rightarrow$ `cons`

Construye un nuevo par dado sus dos valores.

hd : `cons` $\rightarrow$ `any`

Dado un par, obtiene su primer elemento. Lanza un error si el valor no es un par.

tl : `cons` → `any`

Dado un par, obtiene su segundo elemento. Lanza un error si el valor no es un par.

B.1.8. Funciones para manejo de *streams*

Los *streams* en Kλ pueden ser: archivos, que uno abre utilizando la ruta del archivo, o la salida o entrada estándar del programa. A su vez, éstos pueden ser de entrada (reciben bytes) o salida (envían bytes).

write − byte : `number` → `stream` → `number`

Escribe un byte en el *stream* dado. Lanza un error si no es un *stream* al que se le puedan enviar datos (función utilizada: `io:fwrite/3`).

read − byte : `stream` → `number`

Lee un byte del *stream* dado. Retorna `-1` si se llega al final del *stream*. Lanza un error si no es un *stream* del que se le puedan leer datos (función utilizada: `io:get_chars/3`).

open : `string` → `symbol` → `stream`

Abre el archivo ubicado en la ruta dada. Se abre como un *stream* de entrada si se envía como segundo parámetro el símbolo `in` y como *stream* de salida si se envía `out` (función utilizada: `file:open/2`).

close : `stream` → `empty`

Cierra el *stream* de datos. Retorna la lista vacía (función utilizada: `file:close/1`).

B.1.9. Otras funciones

intern : `string` → `symbol`

Dado una cadena de caracteres, retorna su representación como símbolo (función utilizada: `erlang:list_to_atom/1`).

= : `any` → `any` → `boolean`

Dado dos valores retorna `true` si son equivalentes, `false` si no. En el caso de los números, cadenas de caracteres, símbolos y pares, la igualdad está dada

si sus valores son estructuralmente iguales. En el caso de los vectores, *streams* y funciones la igualdad está dada si son idénticos.

La igualdad en Erlang utilizando el operador infijo `=:=` se comporta de la misma manera, por lo que se utilizó para implementar esta función.

$eval - kl$: `any` $\rightarrow$ `any`

Esta función recibe código Kλ como una estructura de datos y lo evalúa. Su implementación se detalla en la sección 5.3.1.

$get - time$: `symbol` $\rightarrow$ `number`

Si el símbolo dado es `run`, se retorna el número de segundos que pasaron desde que se inició el programa.

Si el símbolo dado es `unix`, se retorna la hora en segundos desde el 1ero de Enero de 1970 a las 00:00:00.

type : `any` $\rightarrow$ `symbol` $\rightarrow$ `any`

"Etiqueta" el primer parámetro con un *type hint* que se utiliza para mejorar la eficiencia al llamar a *eval-kl*. Esta función puede ignorar el segundo parámetro y simplemente retornar el primero. Para la primera versión del puerto no se realizó ninguna optimización, por lo que no se agrega ninguna etiqueta a la expresión.

B.2. Símbolos globales predefinidos

Los símbolos globales (y sus tipos) que deben estar predefinidos son:

$*stinput*$: `stream`	La entrada estándar
$*stoutput*$: `stream`	La salida estándar
$*home-directory*$: `string`	La ruta absoluta del directorio "home" del usuario
$*language*$: `string`	El lenguaje del puerto ("Erlang")
$*implementation*$: `string`	La plataforma del puerto ("Erlang/OTP")
$*release*$: `string`	La versión de la plataforma ("20")
$*os*$: `string`	El sistema operativo sobre el que está corriendo
$*porters*$: `string`	Los nombres de los autores del puerto