Contents

1 Task?

1.1 TODO Read the book

1.2 TODO Choose optimizations and extensions

1.3 TODO Give a brief overview of all this opts.

2 Introduction

Vamos a ver una serie de optimizaciones extras que se pueden realizar sobre el compilador. Es lo que van a encontrar en los capítulos que siguen al compilador en sí.

2.1 Garbage Collection

Records allocados en el Heap que ya no pueden ser alcanzados de ninguna forma son considerados **basura**. Idealmente la memoria ocupada con **basura** debería ser reclamada para poder ser utilizada para allocar otros records. Éste proceso es llamado *Garbage Collection* y es un procedimiento realizado por el *runtime* y **no** por el compilador.

Como no es posible discernir directamente cuando una records está vivo (como vieron en la parte de liveness), vamos a tener que diseñar un mecanismo para saber cuando un records es *alcanzable* (reachable). Y la idea es buscar los records que **no son alcanzables** y mantener los que si lo son. El

enfoque es conservador, es decir, vamos a eliminar los que estamos seguros que no se van usar, y mantener los que podrían llegar a usarse.

Un ejemplo sería:

```
let
   type rec = {name : string, id : int}
   // Creamos un record a1
   var a1 : rec := rec {name = "Martin", id = 1}
   // Creamos un record a2
   var a2 : rec := rec {name = "Guillermo", id = 2}
in
// Lo que significa que al comenzar la ejecución ya tendremos en
// memoria a a1 y a2.
   ...
   a1 := rec {name "Martin", id = 89014}
// Claramente a partir de éste punto, si no guardamos la dirección
// del record al que apuntaba a1 antes, perdemos la posibilidad de
// alcanzarlo. Y por ende pasa a ser basura.
   ...
end
```

Hay diferentes técnicas para realizar este proceso:

- Mark and Sweep: Variables y records en memoria forman un grafo direccionado. Podemos pensar que los records alcanzables son aquellos que tienen un camino desde las raíces, las variables. Primero hay un proceso como DFS que marca todos los records que son alcanzados, y luego una etapa de limpieza que reclama todos aquellos que no están marcados (desmarcando los que están marcados preparando para la siguiente iteración). La memoria que se pidió no se libera, sino que se trata de reusar.

- Reference Counts: Se puede llevar cuenta (metadata sobre el records) cuantos punteros/identificadores apuntan a dicha memoria. Cuando el contador llega a 0, se asume inalcanzable.

- Copying Collection: Similar a M/S podemos recorrer todo el grafo de memoria, copiando en otro lugar de memoria los datos que son alcanzados, permitiendo además compactar la memoria. El lugar de memoria donde estaban allocados los records se llama *from-space* y el lugar destino se llama *to-space*. Luego de copiar la memoria, from-space, es basura.

El espacio actual *from-space* contiene además un puntero *next* que indica el

siguiente espacio de memoria, y un limite conocido *limit*. Cuando next alcanza limit, un proceso de GC es disparado, limpiando y compactando la memoria.

- Generational Collection:

Se basa en la idea de que es más probable que los objetos más jóvenes son más probables a ser basura que los objetos que vienen sobreviviendo otras etapas de GC. Así que el heap es dividido en estratos ordenados. El proceso de GC es en orden (y se usan otros métodos).

- Incremental Collection: El objetivo de esta técnica es no cortar la ejecución del programa, y busca ir limpiando el heap a medida que se ejecuta el programa. Se presentan varios algoritmos, basados en marcar los nodos de la memoria representada como un grafo de memorias alcanzables.

Es posible también, ya que el runtime del compilador Tiger está implementado en *C* utilizar una librería que implemente alguna de estas técnicas. Boehm-Demers-Weiser conservative garbage collector.

Garbage Collection es esencial para lenguajes Functional y lenguajes que no hacen uso explicito de la memoria. Básicamente si no queremos manipular directamente la memoria como hacemos en C, vamos a necesitar un GC. Por eso es una técnica critica en la construcción del compilador.

GHC usa un GC Generacional, abusando de la propiedad de *imutabilidad* de los datos. La memoria usada para crear valores en cierto tiempo **no puede hacer referencia a valores creados luego de ese momento** Más información aquí.

Erlang utiliza un proceso mixto, cada proceso en Erlang realiza su propio GC Generacional, pero globalmente se utiliza un *Reference Counter*. Más información aquí

Rust tiene GC? No! Manejo de memoria Manual, como en C. Go utiliza un GC Incremental concurrente *tricolor* con Mark and Sweep.

2.2 Objects-Oriented Languages

La noción de objetos se basan en la idea de ocultación de información, un objeto no revela directamente los datos que tiene, sino que estos son accedidos mediantes métodos. Además debemos implementar nociones de *clases*,

herencia y *extensión*. Es la primera extensión que hará modificaciones principalmente a las primeras etapas del compilador. Se deberá modificar el parser para introducir la noción de *método* y *clases*, *self*, y la creación de objetos de cierta clase *new*. Métodos son tratados básicamente como funciones (en bajo nivel, etiquetas de assembler).

2.3 Functional Programming

Todo lo que está bien. Ver a la programación como la matemática. Funcionales puros, las funciones son funciones, y pasamos a empujar un pensamiento ecuacional sobre la programación. Obtenemos *transparencia referencial*. Además se asume la posibilidad de que las funciones se tomen como argumento o se devuelvan como resultado, llamadas funciones de *alto orden*.

En el libro se presentan 3 posibles enfoques:

- Fun-Tiger: Tiger + Funciones de Alto Orden Se presenta cómo modificar la construcción de los tipos para aceptar el constructor de tipos (->).

Usando la siguiente regla de construcción:

```
ty -> ty ~> ty
   -> (ty{, ty}) ~> ty
   -> () ~> ty
```

Debido a que las funciones pasan a ser valores manipulables, es necesario introducir el concepto de **clausura**. Una clausura representa un llamado a función con información adicional del entorno. En general se representa como una estructura que contiene la dirección de memoria donde está alojada la función más información de cómo acceder a las variables que no son locales a ella. Llevándolo a nuestro *Tiger*, actualmente la referencia a la función (el label) más el *static link* es una clausura.

- Pure-Tiger: Fun-Tiger + Puro. Lenguaje funcional puro con evaluación estricta.

Aquí la extensión principal es el concepto de *immutabilidad* de los valores. Es decir, en Pure-Tiger no es posible realizar ninguna de las siguientes acciones:

- Asignar variables (luego de haberles asignado un valor inicial)
- Mutar campos de los records

- Llamar a funciones externas que produzcan un *efecto* visible: *print*, *flush*, *getchar*, *exit*.

Parece que entonces no podemos hacer nada. Para solucionar las dos primeras restricciones se sigue con una idea de *producir nuevos valores en vez de mutar los que ya se tenían*. Pero ahora, **¿cómo sabemos que está pasando algo si no podemos observarlo?**. Una solución es tener una forma de interactuar con **I/O** de forma encapsulada. Gifs IO in Haskell

Esencialmente podemos solucionar este problema usando una forma de *continuation-based* I/O.

```
type answer
type stringConsumer = string -> answer
type cont = () -> answer

function getchar(c : stringConsumer) : answer
function print (s: string, c : cont) : answer
function flut (c : cont) : answer
function exit () : answer
```

Además todo procedimiento ahora es básicamente un programa que interactúa con el mundo exterior, o no hace nada (literal), así que pasa a ser de tipo *answer*.

Construcciones basadas en asignación de variables son eliminadas del lenguaje, while, for, y el operador (;) (Say hello to **monads**!)

Estar en un lenguaje funcional puro permite realizar un montón de optimizaciones que pueden ver en el libro. Como: Inline Expansion, Efficient Tail Recursion

- Lazy-Tiger: Pure-Tiger + Lazy-evaluation.

Evaluación lazy! Cuando se introdujeron las funciones como valores manipulables del lenguaje se introdujo el concepto de *clausura*, ahora se introduce el concepto de *thunk*. Los valores ya no se evaluaran en el momento que son declarados, o pasados como argumentos, sino que se evaluaran cuando sea necesario. Un **thunk** representa la computación requerida para obtener un valor, cuando sea necesario se disparará, se evaluará el valor correspondiente y luego se continúa con la evaluación del programa. Podemos pensarlo como una función que computa el valor cuando se necesite.

Utilizar thunks en vez de valores directamente nos permite pasar a un modelo llamado *call-by-name* pero todavía estamos en un modelo totalmente

lazy. Además de utilizar thunks debemos garantizar que cada thunk se evalúa una única vez en toda la ejecución del programa. Para esto los thunks se equipan con un slot de memorización que permite alojar el resultado obtenido de evaluarlo, así si se requiere nuevamente simplemente se utiliza ese valor.

Optimizaciones Lazy!

Invariant Hoisting:

```
type intfun = int -> int

function f(i : int) : intfun =
  let function g (j : int) = h(i) * j
  in g end
```

Dado que estamos en lenguajes funcional lazy podemos ejecutar una sola vez a la función h!

```
type intfun = int -> int

function f(i : int) : intfun =
  let var hi := h(i)
      function g (j : int) = hi * j
  in g end
```

Notar que en lenguajes estrictos dicha optimización no siempre es una transformación correcta. Supongamos que h(5) no termina nunca, y definimos var a := f(5) pero no la usamos nunca más... Con efectos está claro que tampoco anda bien.

2.4 Polymorphic Types

Se introduce el concepto de Tipos Parametricos, y todos los problemas que lleva hacer eso. No sé cuanto sentido tenga hablar de esto.

- Parametric Polymorphism: Una función es polimórfica si sigue la misma definición independiente del tiempo de alguno de sus argumentos.

  Se presentan dos lenguajes:

  - **Explicitly Typed Polymorphic Language**: donde los tipos se escriben explícitamente. Por ejemplo: type list<e> = {head : e, tail : list<e>}

- **Implicitly Typed Polymotphic Language**: donde los tipos se infieren, y se traduce a Explicitly Typed.

  Implicit Types dan lugar a algoritmos de inferencia de tipos donde el más usado es Hindley-Milner-Damas. La idea es traducir el problema de inferencia y unificación de tipos a sun sistema de ecuaciones y resolverlo.

- Overloading: Presenta diferentes algoritmos para diferentes tipos de argumentos. Por ejemplo, la función de suma suele estar sobrecargada.

2.5 DataFlow Analysis

Tampoco voy a hablar de esto, ya vieron la idea varias veces a través de las etapas de Alocación de Registros, CSE, Dead-code Elimination, Constant Folding.

2.6 Pipelining and Scheduling

No voy a hablar de esto no llegué a prepararlo.

2.7 Memory Hierarchy

No voy a hablar de esto no llegué a prepararlo.