R-313 Análisis de Lenguajes de Programación

Trabajo Práctico Final

Mariano Crosetti C-5994/3

Índice

1. Introducción

En este trabajo se realiza un intérprete de un lenguaje simple con copatrones.

Los *datos finitos* son usualmente representados por *tipos de datos inductivos* y definidos usando *constructores* y los tipos de *datos infinitos* son representados con *tipos de datos coinductivos* y definidos usando *observaciones*.

No será nuevo para el lector el concepto de *pattern matching* para analizar y definir tipos de datos inductivos. En el presente lenguaje introducimos la posibilidad de incorporar *copatrones* en l.h.s. de las definiciones, un concepto dual para analizar y definir tipos de datos coinductivos.

Se analiza brevemente las ventajas de la incorporación de copatrones. Se especifica la semántica estática (tipado) y dinámica (operacional) mencionando propiedades de seguridad (progreso y preservación) presentes en la publicación en la que basamos nuestro trabajo.

Se profundiza en sentidos no explorados por los autores proponiendo un algoritmo polinomial para analizar cubrimiento de copatrones, se hacen algunas salvedades a propiedades deseadas (como el determinismo, la unicidad de formas normales y la estrategia de evaluación) del sistema creado en la publicación que no se analizan en ella, y se crea una implementación concreta (en Haskell) de la sintaxis (parseo) y semánticas estáticas (tipado) y dinámica (evaluación y reducción).

Las últimas secciones contienen un manual de instalación y uso para rápidamente estar usando el intérprete.

1.1. Ejemplo

Veamos más claramente con un ejemplo. Definiremos el tipo de dato infintos $StreamNat$ y $cycleNat : Nat \to StreamNat$ tal que para alguna constante N:

$$cycleNat\ x = \{x, x-1, .., 0, N, N-1, .,0, N, N-1, ..\}$$

Definiremos $StreamNat$ utilizando observaciones sobre una lista infinita:

$$\textbf{Record}\ StreamNat = \{\ head : Nat \mid tail : StreamNat\ \}$$

Las observaciones o *destructores* pueden ser pensadas como funciones que toman el tipo definido y devuelven otro tipo asosciado. *Deconstruyen* el tipo definido. Lo opuesto a los constructores que toman un cierto tipo asosciado y *construyen* el tipo definido.
Esta no será nuestra sintaxis final, es sólo ilustrativa para el ejemplo pero es conceptualmente equivalente a la que usaremos en nuestro lenguaje.
Por ultimo definiremos $cycleNat$ usando copatrones:

$$head\ (cycleNat\ x) = x$$
$$tail\ (cycleNat\ (1 + x)) = cycleNat\ x$$
$$tail\ (cycleNat\ 0) = cycleNat\ N$$

Como podemos ver, permitimos la presencia de destructores en l.h.s. de nuestras ecuaciones de definición.

En el pattern matching ya conocido sólo podíamos aplicar el símbolo a definir a variables y hacer ajuste de patrones sobre estas variables. Vale remarcar que en este paradigma ya teníamos observaciones para ciertos objetos infinitos: las funciones. Con la aplicación $f\ x$ estamos observando el comportamiento de f.

Ahora permitimos una completa combinación de aplicación, aplicación de destructores y análisis de patrones sobre variables del lado izquierda de las ecuaciones.

La inclusión de destructores al lado izquierdo de nuestras definiciones otorga de elegancia adicional al lenguajes, lo hace más completo y simétrico en el sentido de que provisiona del concepto dúal de patrón (copatrón) para el análisis de objetos infinitos.

De modo que no necesitamos construcciones sintácticas como introductores para nuestros tipos *Record* del lado derecho de nuestras ecuaciones: simplemente podemos definir objetos de este tipo valiéndonos de los copatterns. Del mismo modo que no eran necesarias las funciones anónimas (siempre podemos definir las funciones utilizando aplicaciones del lado izquierdo). Por lo tanto omitiremos de nuestro lenguaje éstas construcciones.

El uso de copattern y observaciones para tipos infintios hace, además, que el sistema de reescritura basado en *"reescribir cuando matchea l.h.s."* sea normalizante. Por ejemplo para definir la lista infinita de número de fibonacci sin copatterns ni tipos definidos por observaciones haríamos algo como:

$$fib = cons\ 0\ (cons\ 1\ (zipWith\ (+)\ fib\ (tail\ fib))$$

La expresión anterior generaría una evaluación infinita en sistemas *"reescribir cuando matchea l.h.s."*. En cambio en nuestro lenguaje definiendo $fib : StreamNat$ podríamos hacer:

$$head\ fib = 1$$
$$head\ (tail fib) = 0$$
$$tail\ (tail\ fib) = zipWith\ (+)\ fib\ (tail\ fib)$$

Podemos ver que tanto el término fib como $tail\ fib$ no aparecen en su propio unfolding (cosa que sí pasa en la primera definición).

La propiedad de *strong normalization* es especialmente deseada si luego quisiéramos incorporar los términos de nuestro lenguaje a un sistema de tipos dependiente (sería necesario para el type checking un procedimiento para la igualdad de los términos que aparezcan en los tipos). Es por ello que la representación con tipos de datos coinductivos y su definición por copatrones es un camino a futuro para incorporar éstos términos (que representen datos infinitos como Stream) en un sistema de tipos dependientes.

El lenguaje utilizado está basado en el definido en [1] con pequeñas modificaciones. A continuación damos su sintaxis, reglas de tipado y describimos su semántica operacional.

2. Sintaxis

Damos el Árbol de Sintaxis Abstracta de los tipos, términos y definiciones del lenguaje.

No damos la sintaxis concreta, el parser ha sido implementado utilizando *happy* [4] y la descripción de la grámatica/lexer se puede observar en el archivo $Parser.p$

La categoría Def agrupa las definiciones que admitimos en nuestro programa (en los archivos cargados). Debe haber al menos un salto de línea entre definiciones. Las definiciones pueden estar en cuelquier orden. Pero cada símbolo definido (para el cuál existe una regla de definición) debe estar declarado (debe tene una signatura). Del mismo modo todo símbolo declarado debe tener al menos una regla que lo defina. Además las reglas que refieren a un mismo símbolo deben estar de forma consecutiva en el código (no puede haber una regla que refiera a otro símbolo en el medio).

$$
\begin{array}{lll}
Def & ::= t : T & \textit{signatura o declaración de tipo.} \\
& \mid T_1 = T_2 & \textit{definición de sinónimo de tipo.} \\
& \mid q = t & \textit{regla de definición de términos.}
\end{array}
$$

En nuestro lenguaje hemos incorporado sinónimos de tipos de modo que podamos reutilizar declaraciones y acortar el tamaño de los tipos haciendo referencia a tipos más pequeños. Cada símbolo de tipo libre debe estar definido. No se puede redefinir un símbolo de tipo. Además no deben existir dependencias circulares en el grafo de definiciones de sinónimos de tipos.

A continuación presentamos el AST de los tipos. Hacemos la distinción entre la clase de tipos positivos y negativos. Tienen un comportamiento dual respecto a la relación de tipados de con los términos. Intuitivamente cuando construimos un término lo tipos positivos se introducen (a partir de los tipos de los subtérminos, como cuando aplicamos un constructor o tenemos pares) y los tipos negativos se eliminan (desaparecen y dan lugar a subtipos, como cuando aplicamos una función o destructor).

$$
\begin{array}{lll}
T & ::= X & \textit{variable de tipo} \\
& \mid P & \textit{Tipo positivo} \\
& \mid N & \textit{Tipo negativo}
\end{array}
$$

$$
\begin{array}{lll}
P & ::= () & \textit{Tipo Unit} \\
& \mid (T_1, T_2) & \textit{Tipo par} \\
& \mid Data(X) < c_1\ T_1 \mid c_2\ T_2 \mid .. \mid c_n\ T_n > & \textit{Tipo data}
\end{array}
$$

$$
\begin{array}{lll}
N & ::= () & \textit{Tipo Unit} \\
& \mid T_1 \to T_2 & \textit{Tipo función} \\
& \mid Record(X)\{\ .d_1 : T_1 \mid .d_2 : T_2 \mid .. \mid .d_n : T_n\} & \textit{Tipo record}
\end{array}
$$

Los tipos Record y Data pueden ser recursivos, es por ello que indican una variable de ligadura. La aparición de la variable ligada hará referencia a todo el tipo definido por el Record o Data que la halla ligado. Por ejemplo nuestro tipo

$StreamNat$ lo definiríamos en nuestro lenguaje como:

$StreamNat = Record(X)\{.head : Nat \mid .tail : X\}$

A veces escribiremos $Data(X)D$ o $Record(X)R$ donde R, D son mapeos $Symbol \mapsto Type$ para constructores y destructores respectivamente. Además notaremos D_{c_i} a T_i para el Data y R_{d_i} a T_i para el Record.
Informalmente explicada la semántica de los constructores y destructores es que:

- Si tenemos un $t : T_i[Data(X)D/X]$ podemos construir un $Data(X)D$ haciendo $c_i\ t$
- Si tenemos un $t : Record(X)R$ podemos aplicar un destructor y obtener $.d_i\ t : R_i[Record(X)R/X]$

Cómo debemos trabajar con la substitución de variables ligadas por tipos, para no tener que lidiar con los problemas cómo renombramiento de variables y para trivializar la comparación por igualdad de tipo, trabajamos en nuestra implementación con índices de Brujín para las variables de tipo ligada, conversión que hacemos al momento del parseo.

En la sintaxis concreta la aplicación de destructor, constructor o término tienen la misma precedencia y asoscian a la izquierda. Del mismo modo el operador de construcción de tipos $\rightarrow$ (Tipo función) asoscia a derecha. Podemos romper la precedencia usando paréntesis.

En la sintaxis aparecen identificadores, comienzan con un caracter alfabético (excepto los destructores) y pueden contener dígitos y '_' . Los identificadores pueden ser:

- Variables de tipos: comienzan en mayúscula. Aparecen en la categoría sintáctica de los tipos. ($StreamNat$ por ejemplo)
- Constructores: comienzan en mayúscula. Aparecen en la categoría sintáctica de los términos. ($Succ$ por ejemplo).
- Variables o símbolos: comienzan en minúscula. (f o x)
- Destructores: comienzan con '.'($.head$ o $.tail$)

El AST de los términos:

$t ::= f$	*símbolo definido*
$\mid x$	*variable ligada por patrón*
$\mid ()$	*unit*
$\mid (t_1, t_2)$	*par*
$\mid c\ t$	*aplicación de constructor*
$\mid t_1 t_2$	*aplicación*
$\mid .d\ t$	*aplicación de destructor*
$\mid integer$	*número entero en notación decimal*

Hemos agregado la posibilidad de incluir números enteros como *syntax sugar*. Al momento de parseo serán traducidos al numeral de Church correspondiente al tipo Nat definido con anterioridad[1]:

$\underline{n} = Succ^{(n)}(Zero())$

Una cuestión a observar: no hay eliminación para los tipos positivos (*case*, fst, *snd*), el análisis de subtérminos en tipos positivos ha de hacerse utilizando *pattern matching*.

Del mismo modo no hay eliminación para los tipos negativos (λ expresiones o record expresiones). La definición de términos de tipos negativos ha de hacerse utilizando *copatterns*

El AST de los copatrones:

$q ::= f$	*símbolo a definir*
$\mid q\ p$	*aplicación de copattern*
$\mid .d\ q$	*copattern destructor*

El AST de los patrones:

$p ::= f$	*variable de patrón*
$\mid q\ p$	*unit*
$\mid (p_1, p_2)$	*patrón pair*
$\mid c\ p$	*patrón constructor*

Además los copatterns deben verificar la propiedad de linearidad: cada variable debe aparecer sólo una vez.

Dado un programa no trabajaremos (para dar el tipado y la semántica operacional) con la lista de definiciones (Def) sino que consideraremos que tenemos un conjunto de reglas y uno de signaturas por símbolos definidos. Además no consideraremos los sinónimos de tipos en la formalización ya que el lenguaje original no lo hace y es simplemente sustituir tipos por sinónimos.

$Programa = (\Sigma,\ Rules)$
$\Sigma : Symbol \to Tipo$
$Rules : Symbol \to \{(q, t)\}$

3. Tipado

La relación de tipado depende de un contexto: $\Delta : Symbol \to Type$.

Tendremos además dos juicios de tipado: inferencia y chequeo de tipos.

[1] $Succ^{(n)}$ indica aplicación sucesiva del constructor $Succ$

- $\Delta \vdash t \Rightarrow A$: se infiere el tipo A
- $\Delta \vdash t \Leftarrow A$: t satisface el tipo A

Las reglas de tipado son guiadas por sintaxis. Para eliminaciones e identificadores podemos inferir el tipo. Pero para introducción como constructores sólo podemos chequearlo. Ya que el tipo de los subtérminos es un subtipo y no podemos deducir de ellos el tipo construido.

Gracias al *chequeo bidireccional* obtenemos sobrecarga de constructores.

A diferencia de [1] ubicamos en las reglas de inferencia la constante () y agregamos una regla de inferencia para pares cuando es posible (inferir implica tener inferencias en las precondiciones, por lo tanto precondiciones más fuertes). Nuestro objetivo es extender el poder de la inferencia a los casos en los que es posible.

$$\frac{}{\Delta \vdash f \ \Rightarrow \ \Sigma(f)} \text{ TC-Fun} \qquad \frac{\Delta(x) = A}{\Delta \vdash x \ \Rightarrow \ A} \text{ TC-Var}$$

$$\frac{\Delta \vdash t_1 \Rightarrow A_1 \to A_2 \quad \Delta \vdash t_2 \Leftarrow A_1}{\Delta \vdash t_1 \ t_2 \Rightarrow A_2} \text{ TC-App}$$

$$\frac{\Delta \vdash t \ \Rightarrow \ Record(X)R}{\Delta \vdash .d \ t \ \Rightarrow \ R_{.d}[Record(X)R/X]} \text{ TC-Dest} \qquad \frac{}{\Delta \vdash () \ \Rightarrow \ ()} \text{ TC-Unit}$$

$$\frac{\Delta \vdash t_1 \ \Rightarrow \ T_1 \ \Delta \vdash t_2 \ \Rightarrow \ T_2}{\Delta \vdash (t_1, t_2) \ \Rightarrow \ (T_1, T_2)} \text{ TC-PairInference}$$

No es posible inferir el tipo sólo en el caso de los constructores (al tener sobrecarga podría haber ambigüedad). Tampoco en pares dónde no se puede inferir alguno de sus miembros.

$$\frac{\Delta \vdash t_1 \ \Leftarrow \ T_1 \ \Delta \vdash t_2 \ \Leftarrow \ T_2}{\Delta \vdash (t_1, t_2) \ \Leftarrow \ (T_1, T_2)} \text{ TC-PairCheck}$$

$$\frac{\Delta \vdash t \ \Rightarrow \ A \ A = C}{\Delta \vdash t \ \Leftarrow \ C} \text{ TC-Switch} \qquad \frac{\Delta \vdash t \ \Leftarrow \ D_c[Data(X)D/X]}{\Delta \vdash c \ t \ \Leftarrow \ Data(X)D} \text{ TC-Constr}$$

Las reglas de inferencia y chequeo de tipos son orientadas por sintaxis (se basan en subtérminos) y existe una regla para cada forma de término (par, constructor, destructor, etc) por lo que vale el lema de inversión. Es por ello que son fácil de ver de forma algorítmica. Para la implementación conviene pensar como funciones para discernir las variables de entrada y salida:

$inferir : término \to Contexto \to Tipo$
$chequear : Tipo \to término \to Contexto \to Bool$

Notas del autor: Uno podría pensar a partir de la existencia de reglas como $TC - Switch$ que podríamos trabajar con chequeo de tipos y que éste es una versión más débil que la inferencia (informalmente $inferenciea \Rightarrow chequeo$). Notar que es necesario tener tipado bidireccional y no se podría chequear el tipo en caso de eliminaciones ya que el nuevo tipo a chequear (que será una variable de entrada en *chequear* para un subtérmino) es un supertipo (un tipo estructuralmente más grande). Si bien el caso de aplicación de destructores se podría solucionar no admitiendo sobrecarga de los mismos, la aplicación de términos sigue sin poderse chequear sin inferencia en la precondición.

Los contextos Δ que aparecen en las reglas de tipado asignan tipos a las variables ligadas por los copatrones. En las reglas de tipado de copatrones también especifican como salida un entorno Δ de variables ligadas:

- $\Delta \vdash p \Leftarrow A$: patrón p admite tipo A ligando las variables Δ.

$$\frac{}{x : A \vdash x \ \Leftarrow \ A} \text{ PC-Var} \qquad \frac{\Delta \vdash p \ \Leftarrow \ D_c[Data(X)D/X]}{\Delta \vdash c \ p \ \Leftarrow \ Data(X)D} \text{ PC-Const}$$

$$\frac{}{\emptyset \vdash () \ \Leftarrow \ ()} \text{ PC-Unit} \qquad \frac{\Delta_1 \vdash p_1 \ \Leftarrow \ T_1 \ \Delta \vdash p_2 \ \Leftarrow \ T_2}{\Delta_1, \Delta_2 \vdash (p_1, p_2) \ \Leftarrow \ (T_1, T_2)} \text{ PC-Pair}$$

- $\Delta \vdash q \Rightarrow C$: copatrón q definiendo símbolo de tipo A es de tipo C ligando las variables Δ.

$$\frac{}{\emptyset \vdash f \ \Rightarrow \ \Sigma(f)} \text{ PC-Hole} \qquad \frac{\Delta \vdash t \ \Rightarrow \ Record(X)R}{\Delta \vdash .d \ t \ \Rightarrow \ R_{.d}[Record(X)R/X]} \text{ PC-Dest}$$

$$\frac{\Delta_1 \vdash q \Rightarrow A_1 \to A_2 \quad \Delta_2 \vdash p \Leftarrow A_1}{\Delta_1, \Delta_2 \vdash q \ p \Rightarrow A_2} \text{ PC-App}$$

Las relaciones definidas por las reglas anteriores también son guiadas por sintaxis y vale un lema de inversión. Pensadas de forma algorítima definirían las funciones:

$inferir : Copatrón \to (Contexto, Tipo)$

$chequear : Tipo \to Patrón \to Contexto$

Para que un programa este bien tipado toda regla $(q, t) \in Rules$ debe cumplir:

$$\frac{\Delta \vdash q \Rightarrow T \quad \Delta \vdash t \Leftarrow T}{\vdash (q, t)} \text{ TC-Rule}$$

Algorítmicamente a la hora de implementarlo se traducirá a ver el tipo del patrón q y obtener el contexto Δ que será usado para tipar las variables libre de t y chequear que éste último tenga el mismo tipo que el que se dedujo de q.

3.1. Consistencia y completitud del tipado

En [1] se tiene un juicio $\Delta \vdash t : T$: "en el contexto Δ el término t se le puede asignar el tipo A". Esta relación tal y como la define no es algorítmica (para eliminaciones 'adivina' el tipo de los subtérminos y para el caso del constructores conoce el tipo $Data$ que se está creando). Se usa como la definición de 't tiene tipo T'. Nosotros trabajaremos con inferencia y chequeo ya que nos interesa implementar el lenguaje. Se verifica:
Type soundness

- $\Delta \vdash t \ \Rightarrow \ T$, luego $\Delta \vdash t : T$.
- $\Delta \vdash t \ \Leftarrow \ T$, luego $\Delta \vdash t : T$.

Type completeness Si $\Delta \vdash t : T$ luego,

- $\Delta \vdash t \ \Leftarrow \ T$
- Si T es tipo Negativo $\Delta \vdash t \ \Rightarrow \ T$.

4. Semántica operacional

La semántica operacional definida en [1] consiste en el sistema de reescritura *"reescribir cuando l.h.s. matchee"*: si $(q, u) \in Rules$ y q matchea con t podemos contraer $t \mapsto q[\sigma]$ donde σ es la substitución de variables por términos que surge del matcheo.

4.1. Relación de matcheo

Para ello definimos las relaciones de matcheos:

- $t =^? p \searrow \sigma$: el término t matchea con el patrón p ligando las variables a términos según la substitución $\sigma : Symbol \mapsto Términos$

$$\frac{}{t =^? x \searrow \{t/x\}} \text{ PM-Var} \qquad \frac{t =^? p \searrow \sigma}{c\ t =^? c\ p \searrow \sigma} \text{ PM-Constr}$$

$$\frac{t_1 =^? p_1 \searrow \sigma_1 \ \ t_2 =^? p_2 \searrow \sigma_2}{t_1\ t_2 =^? (p_1, p_2) \searrow \sigma_1, \sigma_2} \text{ PM-Pair}$$

- $t =^? q \searrow \sigma$: el término t matchea con el copatrón q ligando las variables a términos según la substitución $\sigma : Symbol \mapsto Términos$

$$\frac{}{f =^? f \searrow \emptyset} \text{ PM-Hole} \qquad \frac{t =^? q \searrow \sigma}{.d\ t =^? .d\ q \searrow \sigma} \text{ PM-Dest}$$

$$\frac{t_1 =^? q \searrow \sigma_1 \ \ t_2 =^? p \searrow \sigma_2}{t_1\ t_2 =^? q\ p \searrow \sigma_1, \sigma_2} \text{ PM-App}$$

Es fácil ver que también podemos verlas como funcionas algorítmicas:

$matchPattern : Término \to Patrón \to Substitución(Symbol \mapsto Término)$
$matchCopattern : Término \to Copatrón \to Substitución(Symbol \mapsto Término)$

4.2. Contracción

Estamos en condiciones de definir la contracciónz (que será nuestra única regla de computación):

$$\frac{t =^? q \searrow \sigma}{t \mapsto q[\sigma]} \ (q, u) \in Rules$$

El término t que se contrae constituye el *rédex* de nuestro lenguaje. Un paso de reducción $t_1 \longrightarrow t_2$ consistir en contraer algún rédex de t_1 obteniendo t_2. Hemos dado la regla de contracción que es la única de computación, no listaremos reglas de congruencia.

4.3. Contextos de evaluación

Se define una nueva categoría sintáctica, los *contextos de evaluación*, una generalización de copatrones que admite términos arbitrarios en vez patrones en sus argumentos.

$E ::= f$	*símbolo a definido*
$\mid E\ t$	*aplicación*
$\mid .d\ E$	*destructor*

Informalmente podrían verse como los términos que podrían matchearse con un copatrón (términos con un símbolo definido en su cabeza).

Notas del autor: En [1] las definiciones están invertidas y se define primero los contextos de evaluación y luego la relación de matcheo con copatrones restringiéndonos sólo a contextos de evaluación. Observar que sólo matchearán términos con copatrones si son contextos de evaluación.

4.4. Valores

Definimos los valores del lenguaje con el juicio $\Delta \vdash_v e : A$, la expresión e es un valor de tipo A en el contexto Δ de misma forma que en la publicación [1]. **Obsérvese: que un término sea considerada valor depende de su tipo**.

$$\frac{\Delta \vdash x : A}{\Delta \vdash_v x : A}\ \text{V-Var} \qquad \frac{\Delta \vdash_v v : D_c[Data(X)D/X]}{\Delta \vdash_v c\ v : Data(X)D}\ \text{V-Const}$$

$$\frac{}{\Delta \vdash_v () : ()}\ \text{V-Unit} \qquad \frac{\Delta \vdash_v v_1 : T_1\ \Delta \vdash_v v_2 : T_2}{\Delta \vdash_v (v_1, v_2) : (T_1, T_2)}\ \text{V-Pair} \qquad \frac{\Delta \vdash_v v : N}{\Delta \vdash v : N}\ \text{V-Neg}$$

Notas del autor: En [1] no se propone una estrategia de evaluación clara ya que no se dan reglas de congruencia. Tampoco es claro que en el sistema propuesto exista una (única) forma normal o valor dado un término. No sólo no se nombran propiedades como la propiedad diamante (que garantizan la unicidad de formas normales en sistemas dónde no hay determinismo) sino que existen casos patológicos como el del párrafo siguiente donde valores pueden seguir evaluándose.

Es curioso observar que según la definición de valor y relación semántica planteada el término $t = .tail\ (.tail\ fib\ )$ es un valor ($\Delta \vdash_v t : StreamNat$ por V-Neg) y además $t \longrightarrow zipWith\ (+)\ fib\ (tail\ fib)$ (ya que t es un rédex por matchear con una regla). Uno podría erróneamente creer a primera vista que éstos casos patológicos se limitan a término de tipos negativos (que sean valores por aplicación de la regla V-Neg). Ésto no es cierto porque podría tratarse de valores que *contengan subvalores* de esta clase, como (t, t) o $c\ t$.

Más allá de que creemos que la unicidad de los valores es una característica fundamental ya que es la base para definir una igualdad semántica (dos términos son iguales si computan al mismo valor), para nuestra implementación algorítmica necesitábamos romper con la ambigüedad de saber *hasta cuándo* computábamos. Es decir: reducir un término siempre que podamos, o hasta que satisfaga que el término sea un valor (según la definición).

La primera propuesta (reducir siempre que podamos) es la opción más sencilla: no necesitamos preocuparnos en la implementación por el concepto de valor. Tampoco por definir una estrategia de evaluación. Simplemente tenemos que contraer un término si es un rédex e implementar otra función que reduzca un término si es rédex o si alguno de sus subtérminos lo es. Ésta fue la estrategia implementada en `BruteEval.hs`. No es la usada por defecto pero puede ser usada por medio del comando `:bruteval`.

Por el contrario la reducción usada por defecto sigue la línea de la segunda propuesta. Definimos una categoría sintáctica de los valores (un AST), para lo cuál fue necesario hacer un análisis pertinente (ya que no se da un AST de valores sino reglas de inferencia de la relación "es valor"). Y hemos hecho una función que transforma Términos en Valores reduciéndolo con una estrategia *call by value* (cómo se propone en [1]) deteniéndonos cuando el término a reducir satisface ser un valor.

Como digimos en las notas anteriores, vemos que a diferencia de cuando analizamos lenguajes en clases o en los trabajos prácticos de la materia, donde los valores constituían una categoría sintáctica y podíamos definir el subconjunto de términos que considerábamos valores con un AST (como hacíamos con los términos), ahora debido a que *ser valor* es dependiente del tipo hemos definido usando una relación definida con reglas de inferencia.

Con éstas reglas no es inmediato construir un lema de inversión que nos permita dado un término saber si es valor analizando su estructura sintáctica

porque si la última regla aplicada es V-Neg, no sabemos la forma de v pero podemos hacerlo con un pequeño razonamiento (análogo a los análisis que se hacen en los lema 9, 10, 11 y 14 de [1]).

Sea v un valor cerrado ($\vdash_v v : T$):

- $v = ()$ si $T = ()$.
- $v = (v_1, v_2)$ si $T = (T_1, T_2)$ para v_1, v_2 valores.
- $v = c\ v'$ si $T = Data(X)D$ para v' valor.
- $v = E$ si T es negativo. Siendo E un *contexto de evaluación* como definimos.

Esto nos permite crear un AST para valores:

$$
\begin{array}{lr}
v\ ::= () & \textit{valor Unit} \\
\quad \mid (v_1, v_2) & \textit{valor Pair} \\
\quad \mid c\ v' & \textit{valor Constructor} \\
\quad \mid E & \textit{valor Contexto de evaluación con tipo Negativo}
\end{array}
$$

Hay que hacer la salvedad que si bien éste será el AST que usaremos para representar los valores (que implementaremos como un tipo `data` en Haskell) no todo E (*contexto de evaluación*) es un valor, sino que sólo si tiene tipo negativo.

Notas del autor: Definiremos que un contexto de evaluación E es subcontexto de otro E' si y sólo sí se cumple alguna de las condiciones:

- $E = E'$.
- $E = .d\ E''$ y E' es un subcontexto de E''.
- $E = E''t$ y E' es un subcontexto de E''.

Diremos que E' es un subcontexto *propio* si, además, $E \neq E'$.

Si E es un contexto con sólo valores, tiene tipo positivo (y si nuestro programa tiene un buen cubrimiento de copatrones) el Teorema 12 nos asegura que existirá un subcontexto E' que matchee con alguna regla de nuestro programa. Por lo tanto tenemos un rédex en E' y podremos reducir E.

5. Seguridad del lenguaje

Los lenguajes seguros son aquellos que no poseen términos atascados: dado un término t o bien es un valor o existe un término t' al cual computa.

El sistema de tipos juega un rol fundamental en ésta propiedad clasificando términos según el tipo de valor al que computan y por ende filtrando atascos.

La seguridad se sustenta, como es usual en dos propiedades:

- **Preservación de tipos en la reducción:** Si $\Delta \vdash t \Leftarrow T$ y $t \longrightarrow t'$, luego $\Delta \vdash t' \Leftarrow T$.

  La demostración se puede encontrar en la sección 4.3 de [1]. Al igual que sucede con el cálculo lambda, dónde también la principal regla de computación implica aplicar una substitución, la demostración se basa escencialmente en un lema que dice que la sustitución *bien tipada* (dónde cada término de cierto tipo entra por variables del mismo tipo) preserva tipos.

- **Progreso:** Si $\vdash e \Leftarrow T$, o bien $\vdash_v e : T$ ó existe e' tal que $e \longrightarrow e'$

6. Implementación de la semántica operacional

En [1] si bien se habla de demostrar progreso para una estrategia de evaluación *call by value* creemos que no se propone ninguna estrategia de evaluación ya que no se proveen reglas de congruencia (que escencialmente son las que dan la pauta de la estrategia de evaluación) y sólo se provee la regla de computación (que rige la contracción).

No obstante el Teorema 12 (en [1]) -que nos dice que todo *Entorno con sólo valores* que tenga tipo positivo matchea con al menos un copatrón de alguna definición- *sugiere* que una implementación tenga una estrategia *call by value*, ya que nos garantiza que si reducimos a valores los argumentos de las aplicaciones, podremos hacer matching con alguna regla si la aplicación es suficientemente larga (tipo positivo).

Así nuestro algoritmo de reducción irá traduciendo en valores los términos de forma trivialmente recursiva cuando se encuentre con constructores, pares o unit (reduciéndo a valores los subtérminos). Cuando nos encontremos con una aplicación de un término o destructor sabemos que es un *Contexto de evaluación* y por *Type completness* (Teorema 2 en [1]) sabemos que si tiene tipo negativo podemos inferirlo. Luego trataremos de inferir el tipo y si es negativo ya estamos ante la prescencia de un valor. De lo contrario por *Matching with a covering pattern* (Teorema 12 en [1]) sabemos que (habiéndo un buen cubrimiento de los símbolos definidos) existe alguna regla que matchee con algún subcontexto si es un *Contexto de evaluación con sólo valores*. Por lo cuál, primero reducimos a valores los términos en posiciones de argumentos de E, llevándolo a ser un *Contexto de evaluación con sólo valores* y tratamos de aplicar alguna regla a algún subcontexto de E.

Algunas salvedades en cuánto a la implementación (`Eval.hs`)

- Si el AST de E tiene profundidad $d \in \mathbb{N}$ existirán d subcontextos. Probar para cada uno si matchea con alguna regla es ineficiente. En cambio hemos convertido la representación de los contextos en una que la vea de adentro hacia afuera (empezando por el símbolo respectivo y teniendo una lista de los términos ó destructores aplicados en lo sucesivo). De este modo

para cada regla vamos recorriendo el contexto y el copatrón partiendo del símbolo y viendo si matchea el subcontexto que tenemos hasta el momento. Ésto es mucho más eficiente que probar para cada subcontexto.

- Cuando transformamos E en un *entorno con sólo valores* estamos implementando una una estrategia *call by value*. Notar que como el entorno matcheado será E' un subcontexto posiblemente propio ($E' \neq E$) estaremos potencialmente evaluando términos que no serán argumentos del rédex que elegiremos a continuación. En `Haskell` las definiciones deben tener igual longitud de argumentos. O sea no podemos hacer algo como:

  $f\ 0\ y = y$
  $f\ x = id$

  Una restricción así simplifica mucho el hecho de qué subcontexto elegir (ya que son todos de misma profundidad) y decidir que términos serán argumentos y deban ser evaluados. Pero es inadmisible en nuestro lenguaje ya que la potencialidad de elegancia en las definiciones que dan los copatrones se verían limitada y no se podrían hacer algunas definiciones (como la de $fibo$).

- Un caso trivial de no determinismo que tiene nuestra semántica es que podría haber dos reglas aplicables para un mismo contexto. Creemos que el algoritmo NP propuesto en [1] para covering excluye ésta posibilidad ya que genera conjuntos de reglas mutuamente excluyentes. No obstante no se meciona ésto en la publicación, ni se habla de determinismo. Nosotros hemos decidido desambiguar la semántica asumiendo un **orden de precedencia de las reglas de definición según el orden que aparecen en el código**.

- Si el contexto elegido como rédex es un E' subcontexto propio de E, luego de aplicada la contracción ($E' \mapsto t$) tendremos un término t adentro del contexto E (que para éste momento será un *contexto con sólo valores*) y donde antes aparecía E' entrará t. Llamaremos por ahora t' a este término ($E \longrightarrow t'$) ya que no estamos seguros si es un contexto de evaluación. Pero por *preservación de tipos en la reducción* (Teorema 4 de [1]) t' seguirá teniendo un tipo Negativo y como hemos visto en nuestro análisis sintáctico de los valores será un entorno de evaluación. Por lo cuál en éste caso evaluamos los argumentos de t (transformándolo en un *contexto con sólo valores*) luego lo juntamos con E (lo ponemos en el lugar dónde aparecía E') y evaluamos este nuevo *contexto de sólo valores*.

  Poner a t dentro de E (en el lugar de E') es trivial con nuestra representación *desde adentro* de los contextos de evaluación y dado que la función de matcheo devuelve el segmento de E que no fue matcheado, (o sea, la que no forma parte de E').

- Si el contexto elegido como rédex es $E' = E$ el t al cual E reduce ($E \longrightarrow t$) podría no ser un contexto de evaluación. En este caso el paso siguiente a evaluar es trivial, hay que continuar (recursivamente) la evaluación de t.

7. Cubrimiento de copatrones

El cubrimiento de copatrones cumple un rol fundamental en [1] ya que es escencial para que no haya términos atascados: es una precondición del *Teorema de progreso*.

A pesar de esto los autores de [1] sólo proponen un algoritmo NP y mencionan que confían en que se pueden adoptar algoritmos de cubrimiento eficientes [2].

Como desarrollar un algoritmo nos pareció un desafio sencillo y divertido en la presente implementación incluímos un algoritmo polinomial de cubrimiento (de nuestra completa autoría). No detallaremos una descripción formal más allá del código Haskell (véase `CopatternCovering.hs`) ni demostraremos formalmente que es equivalente al cubrimiento NP planteado en [1].

Nos limitaremos a explicar brevemente los principios que rigen su funcionamiento con un ejemplo sencillo de definición. Supongamos que tenemos las siguiente definiciones (sólo nos interesa l.h.s. de las definiciones).

$f : Nat \to NatList$
$f\ 0 = ..$
$.head\ (\ f\ (x+1)\ ) = ..$
$.head(\ .tail\ (\ f\ (x+1)\ )\ ) = ..$
$.tail(\ .tail\ (\ f\ (x+1)\ )\ ) = ..$

Si definimos $g := f\ 0$ y $h := f\ (x+1)$ podemos pensar que es equivalente a analizar los cubrimientos de las dos definiciones:

$g : Nat$
$g = ..$
$h : NatList$
$.head\ h = ..$
$.head(\ .tail\ h\ ) = ..$
$.tail(\ .tail\ h\ ) = ..$

Si tenemos una expresión dónde aparece $f\ e$, e bien matcheará con 0 o bien con $(x+1)$ por lo tanto es necesario (y claramente suficiente) que g y h presenten *ambas* un buen cubrimiento en sus definiciones para que f esté cubierta.

Esto ahora huele a recursividad! Y mientras g es claramente un caso base (una buena definición es simplemente el símbolo sin ningún copatrón) podemos

eliminar h reemplazando por las definiciones $h_1 := .head\ h$ y $h_2 := .tail\ h$:

$h_1 : Nat$
$h_1 = ..$
$h_2 : NatList$
$.head\ h_2 = ..$
$.tail\ h_2 = ..$

Analogamente a lo que pasaba con f, si tenemos h en un rédex será porque bien le estamos aplicando $.head$ (y estaremos en el caso de h_1) o $.tail$ (y estaremos en el caso de h_2). Luego h tendrá un buen cubrimiento sí y sólo sí lo tienen h_1 y h_2.

Con h_1 ya hemos llegado al caso base, mientras h_2 podrá ser reemplazado nuevamente haciendo definiciones adecuadas de su $.head$ y $.tail$.

No siempre es posible particionar en definiciones disjuntas:

$f : NatList$
$.head\ (\ f\ 0\ ) = ..$
$.head\ (\ f\ (x + 1)\ ) = ..$
$.tail\ (\ f\ x\ ) = ..$

$f\ e$ podría matchear (dependiendo de la forma de e) con el primer y último patrón o con los dos primeros.

Éstos casos y otros de complicación práctica es tratado con elegancia y sencillez en el algoritmo implementeado.

8. Instalación

- Descargar el intérprete

  ```
  $ git clone https://github.com/mariano22/copatterns.git
  ```

- Acceder a la carpeta contenedora

  ```
  $ cd copatterns
  ```

- Instalar usando cabal

  ```
  $ cabal install
  ```

  Cabal instala en `~/.cabal` por lo tanto se debe agregar `~/.cabal/bin` a `\$PATH`:

  ```
  export PATH="$HOME/.cabal/bin:$PATH"
  ```

  Y luego ejecutar con:

```
$ copatterns
Interprete de lenguaje con copatterns.
Escriba :help para recibir ayuda.
CP>
```

Hemos creado también un archivo de lenguajes para nuestra sintaxis de modo que gedit coloree nuestro código. Para ello hay que copiar el `.lang` del repositorio en el directorio de dónde gedit lee los archivos de configuración de cada lenguaje. Suele ser suficiente:

```
sudo cp lcp.lang /usr/share/gtksourceview-3.0/language-specs/
```

A veces es necesario cambiar la versión en `gtksourceview-3.0`. Está configurado para colorear los archivos con extensión `.lcp`.

9. Intérprete

El módulo `Interpreter.hs` implementa un intérprete genérico con las funcionalidades mínimas deseadas de un intérprete interactivo. Permite definir un estado interno y funciones para modificarlo según lo que se lea del archivo/consola. Trabajando con el estado interno de manera abstracta.

Con las funciones que en dicho módulo se exportan se podrían haber programado las funciones principales de los intérprete de comandos de los Trabajos Prácticos de la materia de manera muy fácil y consisa. Viendo los factores comunes que éstos tenían hemos decidido hacer el módulo `Interpreter.hs` (que bien podría haber estado en un paquete separado cómo librería para su reutilización en otros proyectos).

9.1. Manual de uso

Los comandos disponibles en todo intérprete son:

`:load <file>`	Cargar un programa desde un archivo
`:reload`	Volver a cargar el último archivo
`:help`	Mostrar esta lista de comandos
`:quit`	Salir del intérprete
`<exp>`	Evaluar un término

Las expresiones a evaluar de nuestro lenguaje serán de la forma *término* : *Tipo* según las sintaxis definidas en las secciones anteriores (necesitamos el tipo de un término para hacer el chequeo de tipos ya que en nuestro sistema no podemos inferir el tipo de ciertos términos).

Los programas cargados no se acumularán. O sea que cuando cargamos un nuevo programa las definiciones del programa cargado (si hay alguno) se descartarán.

Además hemos agregado los comandos:

`:clear`	Borra las definiciones cargadas.
`:show <symbol>`	Dado un símbolo imprime su definición.
`:show`	Sin argumento imprime todas las definiciones.
`:type <term>`	Inferir el tipo de un término si es posible.
`:bruteval <exp>`	Evalúa un término utlizando evaluación bruta[2].

Nos hemos esforzado en que el sistema de error sea lo más claro posible, siempre incluyendo el número de línea que generó el error (aún cuándo ésto haya significado incluir los números de líneas en nuestra representación de los AST). El error devuelto es el primer error que ocurre (no implementamos recuperación para detectar múltiples errores).

10. Organización del código

A continuación una breve introducción de los módulos del intérprete. Todos se encuentran debidamente comentado si se desea leer con más detalle.

- `BruteEval.hs` Implementa la evaluación bruta, *reducir mientras se pueda*.
- `Common.hs` Define los tipos de datos utilizados.
- `Computations.hs` Define los contextos de computación monádicos utilizados.
- `CopatternCovering.hs` Implementa un chequeo de cubrimientos de copatrones.
- `Eval.hs` Implementa la transformación de términos a valores según se describión en la sección de implementación de la semántica operacional.
- `Interpreter.hs` Librería con funcionalidad básica de intérprete genérico.
- `Main.hs` Módulo principal que define el estado y comportamiento particular del intérprete del lenguaje de copatrones.
- `Parse.y` Especifica la gramática en BNF y provee el lexer.
- `PrettyPrintet.hs` Implementa funciones para mostrar de forma legible términos, tipos y definiciones.
- `Setup.hs` Generado por cabal para la instalación.
- `Syntax.hs` Implementa chequeos sintácticos anteriores al sistema de tipos (por ejemplo no dependencia circular en los sinónimos de tipos).
- `TError.hs` Define el tipo que usaremos para nuestros errores y funciones que generan los errores posibles de nuestro intérprete.

[2] evaluar hasta que no haya redex

- `Topsort.hs` Implementa algoritmo de detección de ciclo por ordenamiento topológico (es usada para el chequeo de dependencias circulares en sinónimo de tipos).
- `Typing.hs` Implementa el chequeo bidireccional de tipos.
- `Utils.hs` Define funciones auxiliares comunes a varios módulos.

En la carpeta `/examples` se encuentren códigos de ejemplos:

- `/noterrors` Códigos que deben ser aceptados. `bigTestFiboOk.lcp` define tipos y funciones complejas y es útil para testear la semántica.
- `/syntaxErrors` Algunos errores causados al momento de parsear ó en los chequos sintácticos posteriores.
- `/typeErrors` Códigos con errores de tipado.

11. Conclusiones

Hemos hecho un análisis de la publicación *"Copatterns: Programming Infinite Structures by Observations"*[1]. Hemos visto que los copatrones hacen al lenguaje de definiciones más elegante, completo y simétrico.

El hecho de haber encarado una implementación nos llevó a razonar las relaciones definidas cómo funciones algorítmicas. Además nos hizo analizar con mayor detenimiento la estructura sintáctica de los valores y definir mejor la semántica operacional. También nos llevó a reflexionar sobre cuestiones (como determinismo o unicidad de valores) que no se tocan en la publicación.

Además creemos que hemos hecho un avance proponiendo un chequeo de cubrimiento de copatrones polinomial. Resta formalizar el algoritmo implementado y probar la equivalencia con el cubrimiento de [1]. Si bien por el momento sólo hemos detectado si las definiciones constituyen o no un buen cubrimiento creemos que fácilmente se podría modificar para que construya testigos que prueben fehacientemente que el cubrimiento es insuficiente.

Por otro lado la implementación es útil a la hora de mostrar de forma más visual, dinámica, y con ejemplos concretos las bondades de las definiciones por copatrones.

12. Referencias

[1] Andreas Abel, Brigitte Pientka, David Thibodeau, Anton Setzer *Copatterns: Programming Infinite Structures by Observations.*
`https://www.cs.mcgill.ca/~dthibo1/papers/popl170-abel.pdf`

[2] U. Norell. *Towards a practical programming language based on dependent type theory.*
`http://www.cse.chalmers.se/~ulfn/papers/thesis.pdf`

[3] Miran Lipovača *Learn you a Haskell for great good!.*
`http://learnyouahaskell.com/`

[4] *Happy User Guide.*
`https://www.haskell.org/happy/doc/html/index.html`

[5] *Haskell Documentation.*
`https://www.haskell.org/documentation/`

[6] *Hoogle.*
`https://hoogle.haskell.org/`