Universidad Nacional Autónoma de México
Facultad de Ciencias
Lenguajes de Programación

Apéndice B: Tipos de datos abstractos

Karla Ramírez Pulido karla@ciencias.unam.mx
Manuel Soto Romero manu@ciencias.unam.mx
Javier Enríquez Mendoza javierem_94@ciencias.unam.mx

29 de agosto de 2018

Descripción

En esta nota se presentan algunos conceptos elementales que permiten definir tipos de datos abstractos, incluyendo operaciones asociadas a los mismos. Los tipos de datos abstractos son de utilidad para definir árboles de sintaxis abstracta u otros datos que usen los intérpretes de los lenguajes que se revisan en otras notas.

Índice general

Índice de códigos

2.1 Tipos de datos abstractos

El lenguaje de programación Racket, proporciona tipos de datos básicos llamados *primitivos*. Estos tipos de datos pueden ser usados por el programador en la solución de problemas. Cada uno de estos tipos es en sí una abstracción que determina un conjuntos de valores, operaciones asociadas a los mismos y algunas restricciones o propiedades.

Por ejemplo, para el tipo de dato `boolean` se tienen dos posibles valores: `#t` y `#f`. Para éste se encuentran definidas las operaciones `and`, `or` y `not`. Un ejemplo de restricción es que el resultado de aplicar cualquier de sus tres operaciones es cerrado, es decir, se regresan valores booleanos.

Aunque estos tipos de datos básicos pueden usarse para solucionar problemas, hay ocasiones en las que se necesita con tipos de datos particulares que no están definidos en el núcleo del lenguaje. Por ejemplo, se si se necesita trabajar con números positivos, sería más adecuado para el programador utilizar datos de tipo `natural`[1] que trabajar con el tipo `number`.

El concepto de tipo de datos abstracto (TDA) es una generalización que permite al programador definir sus propios tipos de datos a partir de otros dados por el lenguaje de programación, especificando la estructura o los componentes del dato y definiendo así operaciones asociadas al nuevo tipo. Se dice que estos tipos de datos son *abstractos* porque nada en su definición especifica cómo deben implementarse, por lo tanto puede haber más de una implementación para cada TDA.

Tipos de datos abstractos en Racket

Para defirnir un TDA en Racket, la variante `plai` provee la primitiva `define-type` que tiene la siguiente sintaxis:

```
(define-type <nombre>
    [<constructor> (<parámetro> <tipo>?)*]+))
```

Se debe especificar un nombre para el tipo de dato y una lista con los constructores para cada posible valor. Además, para cada constructor se debe especificar una lista de parámetros con su tipo, aunque pueden omitirse.

Por ejemplo, para definir un TDA que represente números naturales, se debe especificar un constructor para el cero y otro para el sucesor de un natural:

```
;; TDA para representar números naturales
;; Se tienen constructores para el cero y el sucesor de un número.
(define-type Natural
     [cero]
     [suc (nat Natural?)])
```

Código 1: *Definición del tipo* `Natural`

[1] El tipo `natural` no está definido como primitivo en Racket.

En este caso el tipo de dato es recursivo, pues para construir otros números naturales (mediante el constructor `suc`) se necesita que el parámetros recibido sea otro número natural. Por ejemplo:

```
> (cero)
(cero)
> (suc (cero))
(suc (cero))
```

La primera llamada utiliza el constructor `cero` que no recibe parámetros, mientras que la segunda construye el sucesor del número natural cero que representa al número 1.

Observación Aunque el constructor cero no recibe parámetros, se debe escribir entre paréntesis para que pueda ser aplicado.

Al momento de definir un tipo mediante la primitiva `define-type`, `plai` genera cuatro funciones adicionales:

1. Un predicado `type?` que recibe un parámetro cualquiera y regresa `#t` cuando dicho parámetro se evalúa a algo de este tipo y `#f` en otro caso. Para el TDA `Natural`, se genera el predicado `natural?`.
2. Para cada constructor, se genera un predicado `constructor?`, que recibe un parámetro cualquiera y regresa `#t` cuando dicho parámetro se evalúa a algo de ese constructor y `#f` en otro caso. Para el TDA `Natural`, se generan los predicados `cero?` y `suc?`.
3. Para cada parámetro, de cada constructor, se crea una función para acceder al valor de dicho parámetro que tendrá el nombre `constructor-parametro`. Para el TDA `Natural`, se genera la función `suc-nat`.
4. Para cada parámetro, de cada constructor, se crea una función para modificar el valor de dicho parámetro que tiene el nombre `set-constructor-parametro!`. Para el TDA `Natural`, se genera la función `set-suc-nat!`.

Algunos ejemplos:

```
> (define n1 (suc (cero)))
> (cero? n1)
#f
> (suc? n1)
#t
> (suc-nat n1)
(cero)
> (set-suc-nat! n1 (suc (cero)))
> n1
(suc (suc (cero)))
```

Operaciones asociadas a un TDA

Para definir operaciones asociadas a un TDA hay que definir funciones que usen el tipo de dato definido. A manera de ejemplo se definen dos funciones: una para sumar números naturales y otra para multiplicarlos.

```
;; Función que suma dos números naturales.
;; suma: Natural Natural -> Natural
(define (suma n1 n2)
     (if (cero? n1)
         n2
         (suc (suma (suc-nat n1) n2))))
```

Código 2: *Suma de números naturales*

```
;; Función que multiplica dos números naturales.
;; multiplica: Natural Natural -> Natural
(define (multiplica n1 n2)
    (if (cero? n1)
        (cero)
        (suma n2 (multiplica (suc-nat n1) n2))))
```

Código 3: *Multiplicación de números naturales*

En cada código se hace uso de las funciones que genera la variante `plai` al definir un nuevo tipo de dato. Sin embargo, también es posible usar el mecanismo de apareamiento de patrones.

2.2 Primitivas de Racket

Además de la primitiva `match` existen otras formas de aplicar la técnica de apareamiento de patrones. El uso de estas primitivas es decisión del usuario y del tipo de problema que se está resolviendo. A manera de ejemplo se define un TDA para trabajar con árboles binarios y se definen funciones para recorrer sus elementos usando apareamiento de patrones de dos maneras distintas.

```
;; TDA para representar árboles binarios.
;; Se tienen constructores para el árbol vacío y el árbol con
;; subárboles izquierdo y derecho.
(define-type ArbolB
    [void]
    [mkt (elem any?) (subi ArbolB?) (subd ArbolB?)])
```

Código 4: *Definición del tipo* `ArbolB`

El predicado any? se define como sigue y permite construir estructuras de datos que almacenen datos de tipo genérico al devolver #t con cualquier valor que recibe.

```
;; Función que devuelve #t sin importar el parámetro recibido.
;; any?: any -> boolean
(define (any? a) #t)
```

Código 5: *Predicado* any?

La primitiva type-case

La sintaxis de esta primitiva es la siguiente:

```
(type-case <tipo> <expresión>
    [<constructor> (<parámetro>*) <resultado>]+)
```

Por ejemplo, la siguiente función regresa una lista con los elementos de un árbol en *preorden*.

```
;; Función que obtiene los elementos de un árbol en preorden.
;; preorden: ArbolB -> (listof any)
(define (preorden ab)
    (type-case ArbolB ab
        [void () empty]
        [mkt (elem subi subd)
            (append (list elem) (preorden subi) (preorden subd))]))
```

Código 6: *Función preorden usando* type-case

Para cazar los patrones se consideran tres componentes: (1) el nombre del constructor que se está cazando, (2) los parámetros que recibe el constructor y (3) el valor a regresar si se caza ese patrón.

La primitiva match

La sintaxis de esta primitiva es la siguiente:

```
(match <expresion>
    [<patrón> <resultado>]+)
```

Por ejemplo, la siguiente función regresa una lista con los elementos de un árbol de orden (*inorden*) y otra que los muestra en *postorden*.

```
;; Función que obtiene los elementos de un árbol en inorden
;; inorden: ArbolB -> (listof any)
(define (inorden ab)
    (match ab
        [(void) empty]
        [(mkt elem subi subd)
            (append (inorden subi) (list elem) (inorden subd))]))
```

Código 7: *Función inorden usando* `match`

```
;; Función que obtiene los elementos de un árbol en postorden
;; postorden: ArbolB -> (listof any)
(define (postorden ab)
    (match ab
        [(void) empty]
        [(mkt elem subi subd)
            (append
                (postorden subi)
                (postorden subd)
                (list elem))]))
```

Código 8: *Función postorden usando* `match`

Para cazar los patrones consideran dos componentes: (1) la estructura o el patrón que se está casando y (2) el valor a regresar si se caza este patrón.

Existe una tercera primitiva para realizar apareamiento de patrones, llamada `case`, sin embargo, no es posible utilizarla sobre TDA directamente.

2.3 Referencias

[1] Rodrigo Ruiz Murgía, *Manual de prácticas para la asignatura de Lenguajes de Programación*, Reporte de actividad docente, Facultad de Ciencias, 2016.

[2] Eric Tánter, *PREPLAI: Scheme y Programación Funcional*, Primera edición, 2014. Disponible en: [http://users.dcc.uchile.cl/~etanter/preplai/] (Consultado el 26 de julio de 2017).

[3] Matthias Felleisen, Robert Findler, Matthew Flatt, Shriram Krishnamurthi, *How to Design Programs*, Segunda Edición, The Mit Press, 2017. Disponible en: [http://www.ccs.neu.edu/home/matthias/HtDP2e/] (Consultado el 26 de julio de 2017).

[4] Matthias Felleisen, David Van Horn, Conrad Barski, *Realm Of Racket*, Primera edición, No Starch Press, 2013.

Nota: Esta es una versión preliminar de las notas del curso y están en constante actualización por lo que agradecemos que en caso de detectar algún error o si se desea hacer alguna observación se envíe un correo electrónico a las direcciones `manu@ciencias.unam.mx` y `karla@ciencias.unam.mx` con el asunto `Nota A2: Lenguajes de Programación`.