[Corso A] Secondo Progetto Intermedio

sre990

June 27, 2021

1 Introduzione:

Il progetto prevede la progettazione e realizzazione di una semplice estensione del linguaggio didattico funzionale presentato a lezione, che permetta di manipolare *insiemi*. Un insieme Set è una collezione di valori omogenei, non ordinati, che non contiene duplicati.

Ciascun insieme puo' essere del tipo *Int*, *Float*, *Bool* o *String*.

2 Creazione di un insieme:

In questa sezione, si prende come esempio l'Int, ma e' possibile avere insiemi di tipo Float, String e Bool. La condizione necessaria e' che ogni insieme sia **omogeneo**.

Sono presenti tre costruttori per creare i *Set*:

1. `Empty(t)`

   dove **t** e' il tipo dell'insime.

   Una volta valutata una espressione di questo tipo, otterremo un valore esprimibile del tipo:

   ```
   evT = Set (Int, [])
   ```

   Che corrisponde a un Set vuoto di tipo Int.

2. `Singleton(e)`

   dove **e** e' l'elemento di un singleton.

   Una volta valutata una espressione di questo tipo, otterremo un valore esprimibile del tipo:

   ```
   evT = Set (Int, [Int n])
   ```

   Che corrisponde a un Set con un solo elemento di tipo Int.

3. `Of(t, lst)`

   dove **t** e' il tipo dell'insime e **lst** e' una lista di elementi omogenei.

   Una volta valutata una espressione di questo tipo, otterremo un valore esprimibile del tipo:

   ```
   evT = Set (Int, [Int n0; Int n1; Int n2; Int n3; etc...])
   ```

   Che corrisponde a Set contenente elementi di tipo Int

4. `OfOpt(t, collection)`

   (a) `collection = Empty | Item of exp * collection`

   dove **t** e' il tipo dell'insime e **collection** e' una collezione di elementi omogenei. La collection permette di avere, sintatticamente, il valore *Empty* e un *Item(t,item)*.

   Questa implementazione ci evita di andare a ottenere il valore che ci serve in un altro dominio (quello della List).

   Il risultato della valutazione dell'espressione OfOpt e' esattamente lo stesso di quello di *Of(t,lst)*, con la differenza che questa e' una versione che ci permette di avere ottimizzazioni operando direttamente sull'*AST*.

3 Tipi:

Un *insieme* Set è una collezione di valori omogenei, non ordinati, che non contiene valori duplicati.

Sono definiti dei *typeSet*, cioe' tutti i tipi che un insieme puo' contenere. Ciascun insieme puo' essere del tipo *Int*, *Float*, *Bool* e *String*.

4 Funzioni:

- `get_type (v : evT) : typeSet`

  che prende in input un valore esprimibile e restituisce il tipo corrispondente

- `get_type_set (x : exp) : typeSet`

  che prende un'espressione e restituisce il tipo corrispondente

- `eval_list (t : typeSet, collection : exp list,`

  `acc : evT list, s : evT env) : evT list`

  che viene chiamata dal costruttore *Of(t, collection)* e serve per valutare una lista di espressioni.

- `set_forall, set_exists,set_filter, set_map`

  funzioni helper chiamate in eval per valutare gli operatori di natura "funzionale". Vedere i commenti nel codice del file *interpreter.ml*, che descrivono dettagliatamente il loro comportamento.

  In particolare, per quanto riguarda il *set_map*, ho deciso come scelta di implentazione che il tipo del risultato di ciascun elemento del set risultante dal mapping di una funzione su un set sara' lo **stesso** del predicato. Ovvero, alla Map e' concesso **cambiare** il tipo di un set (cfr. funzione *map_turn_to_one* nel file *tests.ml*). Gli errori vengono genarati **solo** nel caso in cui il primo parametro non sia un predicato e il secondo non sia un set.

  Invece, per quanto riguarda il *set_filter*, controlliamo innanzitutto che il primo paramentro sia un predicato (come nella *set_map*) e alla fine restituiamo l'espressione valutata che ha lo stesso tipo del parametro set. Analogamente alla map, avremo un errore nel caso in cui il primo parametro non sia un predicato e il secondo non sia un set.

  Infine, la *set_forall* e la *set_exists* si accertano che il primo parametro sia un predicato e alla fine restituiscono un booleano. Come nella map e nella filter, avremo un errore nel caso in cui il primo parametro non sia un predicato e il secondo non sia un set.

- varie funzioni helper usate in *eval* per valutare ciascuna espressione.

5 Espressioni:

- `BiggerThan, LessThan, Eq`

  per confrontare due espressioni di tipo Int, Float, String.

- `Concat`

  per concatenare due String

- `Union, Intersection, Difference, Add, Remove,`

  `IsEmpty, IsInside, IsSubSet, GetMax, GetMin`

  che sono le operazioni richieste dalla specifica

- `Head, Length`

  che sono operazioni su Set per ottenere, rispettivamente, l'elemento in testa alla lista e la lunghezza della lista

- `ForAll, Exists, Filter, Map`

  che sono le operazioni di natura "funzionale" richieste dalla specifica. Ciascuna operazione, per poter essere valutata, si serve delle funzioni:

  `set_forall, set_exists, set_filter, set_map`

6 Testing:

I test sono stati progettati in ambiente *REPL*, pertanto - al fine di testare l'interprete - e' necessario incollare prima il codice del file *interpreter.ml*.

Dopo aver valutato il codice dell'interprete, per utilizzare la batteria di test, e' possibile copiare e incollare dal file *tests.ml* tutte le righe fino alla linea tratteggiata.

Oltre quella linea cominciano i test per la gestione degli errori. Dunque, da li' in poi, e' necessario copiare e incollare riga per riga.

Per testare il typechecker statico, il procedimento e' analogo, occorre soltanto utilizzare i file *static_tc.ml* e *static_tc_tests.ml*.