Relazione `Interprete Funzionale` secondo progetto (PR2B)

Matteo Giorgi

Gennaio 2017

In progetto `Interprete Funzionale` implementa in OCaml un interprete di un linguaggio funzionale didattico con scoping statico e type checker dinamico.

La sintassi elementare del linguaggio didattico é stata arricchita inserendo i costruttori di funzioni ricorsive (`Rec`), funzioni unarie e n-arie (`UFun`, `Fun`), tuple di espressioni (`Etup`), pipeline di funzioni (`Pipe`) e iterazione di funzioni (`ManyTimes`).

Ambiente funzionale

L'ambiente utilizzato é una funzione `string->'t` (nel caso specifico `string->eval`) definito nel modulo `Funenv` e corredato delle funzioni

- `emptyenv`: ambiente vuoto con valore di bottom
- `applyenv`: applicazione dell'ambiente ad una stringa
- `bind`: ambiente esteso con un nuovo bind
- `bindlist`: ambiente esteso con una lista di nuovi bind

Tipo `eval` ed estensione della funzione di semantica `sem`

L'idea é stata quella di estendere il tipo `eval` dei valori con il solo costruttore `TupVal` necessario a dare semantica alle tuple; per le altre due espressioni richieste (`Pipe` e `ManyTimes`), l'intenzione é stata quella di valutarle con chiusure di funzioni `Funval`.

La definizione di una `Pipe` equivale a quella di una `UFun` costruita tramite la composizione delle funzioni presenti nella tupla passatale come argomento; analogamente quella di una `ManyTimes` é pari alla definizione di una `UFun` costruita componendo la funzione in argomento con sé stessa `m` volte (altresí non é che una `Pipe` di una tupla composta da `m` volte la stessa funzione).

Per realizzare quanto sopra é stato necessario costruire la funzione `substitute` che ricorsivamente sostituisse la variabile di ciascuna funzione con il corpo della funzione a lei precedente nella pipeline. La difficoltá di questo tipo di implementazione sono i casi di omonimmia tra la variabile della futura funzione finale ed i parametri delle altre funzioni della pipeline (che hanno quindi variabile diffente), come nei TEST_9 e TEST_12:

```
(*TEST_9*)
sem(
    Let("x", Eint 10,
        Let("p", Pipe(Seq(
                        UFun("x", Sum(Den "x", Eint 1)),
                        Seq(UFun("y", Sum(Den "y", Den "x")), Nil))),
            UAppl(Den "p", Eint 100))),
            emptyenv(Unbound)
);;

(*TEST_12*)
sem(
    Let("x", UFun("k", Prod(Den "k", Eint 3)),
        Let("p", Pipe(Seq(
                        UFun("x", Sum(Den "x", Eint 1)),
                        Seq(UFun("y", UAppl(Den "x",Den "y")), Nil))),
            UAppl(Den "p", Eint 10))),
            emptyenv(Unbound)
);;
```

La soluzione é stata quella di sostituire ciascun identificatore omonimo alla variabile della funzione finale con un identificatore-segnalino (`__x`) che nell'ambiente di dichiarazione della `Pipe` sia valutato con lo stesso valore dell'eventuale identificatore omonimo che va a sostitiure.

Conclusioni

L'implementazione scelta permette di dichiarare funzioni come `Pipe` contenenti tuple di `UFun` miste a `Pipe`, `ManyTimes` oltre che funzioni precedentemente allocate, indirizzabili quindi con il proprio identificatore, vedi TEST_11:

```
(*TEST_11*)
sem(
    Let("x", UFun("k", Prod(Den "k", Eint 3)),
        Let("p", Pipe(Seq(
                        UFun("x", Sum(Den "x", Eint 1)),
                        Seq(Den "x", Nil))),
            UAppl(Den "p", Eint 10))),
            emptyenv(Unbound)
);;
```