Compost

September 20, 2023

```
val name_email_map : (string * string) list =
[("Roger Burtonpatel", "roger.burtonpatel@tufts.edu");
 ("Randy Dang", "randy.dang@tufts.edu");
 ("Jasper Geer", "jasper.geer@tufts.edu");
 ("Jackson Warhover", "jackson.warhover@tufts.edu")]
```

1 Introduction

Here, introduce our readers to the Compost language and give a brief overview on how it may be used.

2 Compost Features

2.1 Uniqueness types

2.2 Hindley-Milner Type Inference

2.3 Algebraic Data Types with Top-Level Pattern Matching

List out 2-3 interesting features of the language.

3 Code Example

```
(define-datatype list   ;; example datatype definition of a list of ints
    [cons   ;; the ‘cons’ tag is both the name of the function that introduces
            ;; a value of this variant and the tag used during pattern matching
        (: car int)     ;; the tag is followed by type annotations of the form
                        ;; (: <field-name> <type>)
        (: cdr list)
    ]
    [nil]
)

;; ‘concat’ takes in two lists of integers ‘xxs’ and ‘ys’ and returns another
;; list containing all elements in ‘xxs’ followed by all elements in ‘ys’.
(: concat (-> (list list) list))    ;; a top-level type annotation
                                    ;; giving a type for ‘concat’,
                                    ;; which is a function that takes in
                                    ;; two lists and returns a list
(define concat (xxs ys)
    (match xxs  ;; a pattern match. ‘xxs’ is now considered out-of-scope
        [(cons x xs)    ;; a pattern. this deallocates ‘xxs’’s top level ‘cons’
                        ;; and binds ‘x’ to its ‘car’ and ‘xs’ to its ‘cdr’
            (cons x (concat xs ys)) ;; ‘cons’ is used to introduce a ‘(list a)’,
        ]                           ;; corresponding to an allocation
        [(nil) ys]      ;; in this branch, the ‘nil’ is deallocated,
                        ;; resulting in the complete destruction of ‘xxs’
    )
)

;; ‘filterge’ takes in an integer ‘n’ and a list of integers ‘xxs’ and returns
;; a list of all elements in ‘xxs’ that are greater than or equal to ‘n’.
(: filterge (-> (int list) list)    ;; a top-level type annotation giving
                                    ;; a type for ‘filterge’
(define filterge (n xxs)
    (match xxs  ;; a pattern match. ‘xxs’ is now considered out-of-scope
        [(cons x xs)    ;; deallocates ‘xxs’ top level ‘cons’ and binds
                        ;; ‘x’ to its ‘car’ and ‘xs’ to its ‘cdr’
            (if (>= x n)    ;; ‘x’ is a primitive value,
                            ;; so does NOT go out-of-scope
                (cons x (filterge n xs))    ;; we allocate a ‘cons’
                (filterge n xs)
            )
        ]
        [(nil) (nil)]   ;; in this match, we deallocate a ‘nil’ but
                        ;; allocate a new ‘nil’
    )
```

```
)

;; ‘filterlt’ takes in an integer ‘n’ and list of integers ‘xxs’ and returns
;; a list of all elements in ‘xxs’ that are less than ‘n’.
(: filterlt (-> (int list) list)    ;; a top-level type annotation giving
                                    ;; a type for ‘filterlt’
(define filterlt (n xxs)
    (match xxs  ;; a pattern match. ‘xxs’ is now considered out-of-scope
        [(cons x xs)    ;; deallocates ‘xxs’ top level ‘cons’ and binds
                        ;; ‘x’ to its ‘car’ and ‘xs’ to its ‘cdr’
            (if (< x n)     ;; ‘x’ is a primitive value,
                            ;; so does NOT go out-of-scope
                (cons x (filterlt n xs))    ;; we allocate a ‘cons’
                (filterlt n xs)
            )
        ]
        [(nil) (nil)]   ;; in this match, we deallocate a ‘nil’ but
                        ;; allocate a new ‘nil’
    )
)

;; ‘quicksort’ takes in a list of integers ‘xxs’ and returns another list
;; with the same elements as ‘xxs’, but sorted in ascending order.
(: quicksort (-> list list))
(define quicksort (xxs)
    (match xxs  ;; match expression moves ‘xxs’ goes out of scope
        [(nil)  ;; in this match, we deallocate a ‘nil’
            (nil)   ;; here, we allocate a new ‘nil’
        ]
        [(cons x xs)    ;; in this match, we deallocate a ‘cons’,
                        ;; bind its ‘car’ to ‘x’, and bind its ‘cdr’
                        ;; to ‘xs’
            (let*
                (
                    [lesser
                        (filterlt
                            (x (dup xs))  ;; ‘dup’ creates a deep copy of
                                          ;; ‘xs’ so it does NOT go out-of-scope
                        )
                    ]
                    [greater (filterge (x xs))]   ;; ‘xs’ is consumed here and thus
                                                  ;; goes out-of-scope
                )
                (concat
                    (quicksort lesser)
```

```
                    (cons x (quicksort greater))    ;; allocate a 'cons'
                )
            )
        ]
    )
)
```

Note: Some example proposals interleave features with mini-code examples of that feature. We may want to consider that.

4 Additional Sections

Additional sections may include unique challenges for our language, background information, etc.