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

Compost is a general-purpose functional programming language with a uniqueness type system. That is, the type system guarantees that at most one reference to an object exists at any time. Programs in Compost include no explicit memory management and run without the need for a runtime garbage collector. In a manner akin to Rust, we automatically insert allocate and free directives at compile time, guaranteeing memory safety. The uniqueness type system is an interesting paradigm that leads to unique design decisions, and one that we're excited to explore in our ambitious project.

2 Compost Features

2.1 Uniqueness types

Our type system guarantees that only a single reference exists to any object at a time by moving names out of scope once they have been "consumed" in one of a few ways. A name is consumed when it is passed to a function, passed out as a return value, or pattern matched on. Values of primitive types (char, int, bool), along with functions, are exempt from these rules.

We imagine that each function has a set of names that it is "responsible for". A function is initially responsible for all of its parameters, and becomes responsible for any fresh names that it binds, either in pattern matches or let expressions. A function loses responsibility for any name that is consumed. Because Compost is a language of expressions and lacks (at the moment) any kind of sequencing expression, we can compute this set at any point in a function's code.

The compiler ensures that deallocation is performed in two scenarios. First, at the end of each of its paths, a function is responsible for freeing the memory associated with all of the names it is still responsible for. Values of primitive types (char, int, bool), along with functions, are exempt from this because they are not heap-allocated. Second, deallocation occurs in a pattern-matching `match` statement. In this case, the top-level data structure is freed (for example, the top-level cons cell), and its children are bound by new names. In both cases, we can deallocate safely because of the invariant supplied by uniqueness types. In addition, the concept of responsibility ensures that all allocated memory will be freed by the end of a program's execution. A function either frees a name's associated memory or hands the responsibility off to another function.

2.2 Hindley-Milner Type Inference

Implementing Hindley-Milner Type Inference allows for users to define let expressions without explicitly stating the types of the bindings, avoiding all sorts of pain in defining and applying type constructors. The compiler would instead infer the types based on how it is used. See lines 77-93 of our code example for an example of a let expression.

2.3 Algebraic Data Types with Top-Level Pattern Matching

With Compost, users can define their own algebraic data types (such as lists and trees) using the `define-datatype` keyword. A name for the datatype is supplied, along with a series of variant definitions. Each variant definition includes a name (the tag) followed by a series of type annotations which name and type its fields. See lines 1 to 9 of our code example for an example.

The tag is used in two ways. First, it is used as the name of a function which takes arguments whose types match the types of the variant's fields and introduces a value of that variant. See line 21 of our code example for a use case. The second is in what we call a "top-level pattern match". This resembles a match case expression in Haskell or ML but is really just branching on the value of the tag. See line 18 of our code example for an example. Each branch of the expression contains a "pattern", a tag followed by names to bind each of its fields, and an expression. Informally, the semantics of the top-level pattern match are that the first branch with a tag matching that of the value being matched on is taken, the names in the pattern are bound to the values of the associated fields of the value, and the expression is evaluated in an environment augmented with the newly bound names.

This system achieves our goal of being able to extract all fields of a value simultaneously without the complexity of a match compiler. A match expression should compile to a switch instruction, with each branch containing a cast and a series of assignments.

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',
        ]                         ;; 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'
                    )
               )
          ]
     )
)
```

4 Reach Goals

4.1 Polymorphism

Our proposal assumes monomorphic types for Compost's type system. This is an intentional choice to minimize complexity of implementation. We would, however, love to enable polymorphism by parameterizing Compost's ADTs. To do so, the language specified until this point would become an intermediate representation, and we would implement a new frontend supporting "polymorphic Compost". A monomorphizing pass would determine every monomorphic instantiation of each function and datatype and create monomorphic copies of them with names prefixed based on type arguments. We would reuse the rest of the compiler downstream from monomorphic Compost.

4.2 Closures

Anonymous functions which capture their environment are a key feature of many functional programming languages and one which we'd like very much to implement in Compost. However, these present a few challenges with our type system. At the moment, the programmer is free to apply a function multiple times without having to invoke `dup`, as they are just function pointers under the hood. Closures, however, must store captures values from the environment, some of which may be heap objects that must be freed. How exactly to handle these is its own challenge and not one that we want to get in the way of implementing the core of Compost.