Pure Garbage: The Compost Language Reference Manual

October 18, 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 statically-typed pure functional programming language with automatic compile-time memory management enabled by its uniqueness type system.

2 Notational Conventions

Code listings will appear in "verbatim" as follows:

```
(define foo ()
  bar)
```

Grammar rules are written in extended Backus-Naur format, as follows:

rule	::=	(*nonterminal* `terminal`)
	|	{ *other-rule* }

Parentheses are concrete syntax, but any pair of balanced parentheses may be freely exchanged for a pair of square brackets. For example, the following two declarations are indistinguishable in the abstract syntax:

```
(val x 1)
```

```
[val x 1]
```

Note that braces are used in a manner akin to the Kleene closure, that is, a term enclosed in brackets may be omitted or arbitrarily repeated.

3 Lexical Conventions

3.1 Whitespace

The following characters are considered as whitespace and, with one exception, ignored during tokenization: spaces, tabs, carriage returns, and newlines.

3.2 Comments

Comments are introduced by the character `;` and terminated by the newline character. Comments are treated as whitespace.

3.3 Literals

literal	::=	*integer-literal*
	|	*symbol-literal*
	|	*boolean-literal*
	|	*unit-literal*

Literals introduce values of Compost's primitive types. All literals are valid expressions.

3.4 Integer Literals

integer-literal	::=	token composed only of digits, possibly prefixed with a `+` or `-`.

The `+` prefix denotes a positive integer and the `-` prefix denotes a negative integer. The characters `1 2 3 4 5 6 7 8 9 0` are considered digits.

3.5 Symbol Literals

symbol-literal	::=	`'`{ *symbol-character* }`'`
symbol-character	::=	any unicode code point other than `'` and the backslash character unless escaped with a backslash.

That is, any sequence of `'`-delimited unicode characters is a valid symbol literal, as long as every instance of `'` or backslash are preceded by a backslash. These escape sequences are replaced by their unescaped counterparts in the introduced symbol value. For example, the following are valid symbol literals:

```
'\'hello, world\''

'\\ is a backslash'

'I exist
on multiple lines'
```

3.6 Other Literals

boolean-literal	::= `true` | `false`
unit-literal	::= `unit`

3.7 Reserved Words

The following tokens are considered reserved:

```
; ( ) [  ] : _ -> if val define datatype use case begin let dup int bool sym unit
```

4 Values

This section describes the kinds of values manipulated by Compost programs.

4.1 Integers

Integer values are 32-bit signed integers with a range of -2,147,483,648 to 2,147,483,647.

4.2 Symbols

Symbol values are interned immutable strings of unicode characters.

4.3 Booleans

Boolean values are either the boolean $true$ or the boolean $false$.

4.4 Unit

Unit values are the value *unit*.

4.5 Variant Values

Variant values are either a constant constructor or a non-constant constructor applied to a series of value arguments. We write an arbitrary constant constructor c as (c) and an arbitrary non-constant constructor d applied to arguments $v_1...v_n$ as $(d\ v_1\ ...\ v_n)$.

Variant constructors are monomorphic, that is, for any constructor c, there exist types $\tau_1...\tau_n$ such that for any application of constructor c to arguments $v_1...v_n$, v_i must have type τ_i for all $i \in 1...n$.

4.6 Functions

Functions in Compost are first-class objects. Function values are mappings from ordered sets of values, to values. That is, a function f, when applied to values $v_1...v_n$, produces a value v_r. Like variant constructors, functions are monomorphic, so the types of $v_1...v_n$ and v_r are fixed.

5 Names

Compost places relatively liberal constraints on the sequences of characters considered valid names.

name	::=	any token that is not an *int-lit*, does not contain whitespace, a `'`, bracket, or parenthesis, and is not a reserved word.

Names are bound to types, values, and variant constructors, and used to refer to them at various points in a program. Names are also used in the *use-declaration* syntactic form to refer to files.

6 Type Expressions

type-expression	::=	*function-type*
	|	*int-type*
	|	*bool-type*
	|	*sym-type*
	|	*unit-type*
	|	*datatype*

6.1 Primitive Types

int-type	::=	`int`
bool-type	::=	`bool`
sym-type	::=	`sym`
unit-type	::=	`unit`

`int` is the type of integer values.

`bool` is the type of boolean values.

`sym` is the type of symbol values.

`unit` is the type of unit values.

6.2 Function Types

function-type	::=	(`->` ({ *type* }) *type*)

$(\rightarrow (\tau_1 \ ... \ \tau_n) \ \tau_r)$ is the type of function values which map ordered sets of values $v_1...v_n$ of types $\tau_1...\tau_n$ to values v_r of type τ_r.

6.3 Datatypes

datatype ::= *name*

Datatypes are the types of variant constructor values. Multiple variant constructors may share the same type. Datatypes and their constructors can be defined by the programmer with the following syntax:

datatype-definition ::= (`datatype` *name* ({ *variant-constructor-definition* }))

variant-constructor-definition ::= (*name* ({ *type-expression* }))

A *name* bound to the new type τ_d appears directly following the `datatype` keyword, and this is followed by a list of variant constructor definitions. Each of these provides a *name* bound to the constructor, c, followed by a list of *type-expression*s $\tau_1 ... \tau_n$ typing its arguments. Given this definition, a variant value $(c\ v_1\ ...\ v_n)$ of type τ_d may be introduced by applying function value c to $v_1 ... v_n$, where the type of v_i is τ_i for all $i \in 1...n$

The placement of a datatype or variant constructor's definition has no bearing on where it can be referenced, introduced, or eliminated. In fact, datatypes may be defined recursively, as in the following example:

```
(datatype int-list
  (
    [cons (int int-list)]
    [nil ()]
  )
)
```

This declaration can be read as: "an `int-list` is either `cons` applied to an `int` and an `int-list`, or `nil` applied to nothing".

7 Type System

Compost is statically typed, with a monomorphic

8 Expressions

expr ::= *literal*
| *name*
| *case-expression*
| *if-expression*
| *begin-expression*
| *apply-expression*
| *let-expression*
| *dup-expression*

Meaningful computation is encoded in Compost as *expr* syntactic forms, or expressions. These appear either as the right-hand side of `val` declarations or as the bodies of functions.

We describe the semantics and typing rules of expressions largely informally but use formal notation to aid conciseness. Expressions are evaluated in an environment ρ mapping names to values. Initially, these environments contain the values and types of all globally bound names (functions, `val`-bound names). $\rho[x \mapsto v]$ is the modified environment ρ in which name x is bound to value v. $\rho[x]$ is the value mapped to by x in ρ.

Certain expressions will "consume" names, though note that only names mapped to variant values are consumed. That is, if a subsection states that "if `e` is a name, it is consumed", this should be interpreted as: "if `e` is a name bound to a" Consumption is defined inductively on the structure of expressions by the following subsections.

8.1 Case Expressions

case-expression ::= (`case` *expr* ({ *case-branch* }))

case-branch ::= (*pattern expr*)

pattern ::= (*name* { *name* | _ })
| _

Values of the form ($c\ v_1\ ..\ v_n$) are eliminated by the *case-expression* syntactic form. Consider a case expression with n branches of the form:

```
(case e
  (
    [(c1 v11 v12 ...) e1]
    ...
    [(cn vn1 vn2 ...) en]
  )
)
```

Suppose evaluation of `e` in environment ρ yields a value $v = (c\ v_i\ ...\ v_m)$. We assert that the type of `e` must be a datatype. Each `c`i where $i \in 1...n$ must be a variant constructor of τ_d. If there exists some branch whose pattern is prefixed by c, we evaluate it in the modified environments Γ', ρ identical to ρ, Γ' but with every name consumed in `e` unbound, and yield the result.

Suppose this branch is the `case-branch` containing the pattern prefixed by variant constructor `c`k. Evaluation proceeds as follows. Let $\tau_1 ... \tau_m$ be the types of c's arguments. We assert that number of names or wildcards `v`ki following `c`k must be precisely m. Evaluation of this branch yields the result of evaluating `e`k in the modified environments $\rho'[\texttt{v}k1 \mapsto v_1, ..., \texttt{v}km \mapsto v_m]$. Note that we do not bind the wildcard symbol in ρ'.

8.2 If Expressions

if-expression ::= (`if` *expr expr expr*)

Consider an if expression of the form:

```
(if e1 e2 e3)
```

Suppose evaluation of `e1` in environments ρ yields a value v. We assert that the type of `e1` must be `bool`. Let

8.3 Begin Expressions

begin-expression ::= (`begin` { *expr* })

8.4 Apply Expressions

apply-expression ::= (*expr* { *expr* })

8.5 Let Expressions

let-expression ::= (`let` ({ *let-binding* }) *expr*)

let-binding ::= (*name expr*)

8.6 Dup Expressions

dup-expression ::= (`dup` *name*)

9 Definitions

Syntactic forms in the *def* category are allowed only at the top level of a Compost program.

def ::= *val-binding*
| *function-definition*
| *datatype-definition*
| *type-annotation*
| *use-declaration*

9.1 Global Bindings

val-binding ::= (`val` *name exp*)

The `val` form introduces an immutable globally scoped binding. Note that the bound name is in scope only in code located below the declaration. For example the programmer may declare a global constant:

```
(val triangle-sides 3)
```

Now, any successive code may use `triangle-sides` in place of the literal `3`.

9.2 Function Definitions

function-definition ::= (`define` *name* ({ *name* }) *exp*)

The `define` form defines a function in the global scope. Any function may be referenced from any location in the program. For example, the following introduces a function that returns the greater of its two arguments:

```
(define max (x y)
  (if (> x y)
      x
      y
  )
)
```

The name directly following the `define` keyword is bound to the function in the global scope, and the list of names within the parentheses are bound, within the function body, to its arguments.

Note that the order of function declarations does not matter. Thus, the programmer may write mutually recursive programs like so:

```
(: is-even (-> (int) bool))
(define is-even (x)
  (if (= x 0)
      true
      (not (is-odd (- x 1)))
  )
)

(: is-odd (-> (int) bool))
(define is-odd (x)
  (if (= x 1)
      true
      (is-even (- x 1))
  )
)
```

That is, despite `is-odd` being declared after `is-even`, `is-even` is in scope within the body of `is-odd`.

9.3 Type Annotations

type-annotation ::= (`:` *name* *type-expression*)

Type annotations constrain the type of globally bound names. In Compost, such names appear in two forms: the names of functions and names bound globally by `val`. Take the definition of function `max` from the previous section. We left out one key point, that this definition cannot stand on its own. Each function definition must have a corresponding type annotation that gives its type. This annotation may appear before or after its corresponding definition, but it must exist. For example, the definition of `max` may be succeeded by the following annotation:

```
(: max (-> (int int) int))
```

That is, `max` is a function that takes two integer arguments and returns an integer. Though the type of a function must be an "arrow" type, `->` may appear more than once in a function's type annotation. For example, consider the following:

```
(: add-or-sub (-> (bool) (-> (int int) int)))
(define add-or-sub (cond)
  (if cond
      +
      -
  )
)
```

Given that `+` and `-` are binary addition and subtraction on integers as defined in the initial basis, `add-or-sub` is a function that, given a boolean argument, returns a functions from pairs of integers to integers. Note that Compost has no support for closures, so we cannot express the full range of higher-order functions.

As mentioned, the other use of a type annotation is to type a `val`-bound name. As with function definitions, all `val` bindings must have an associated type annotation. However, the scoping rules here are more subtle. Although a `val` binding's type annotation may be located before or after the binding itself, the bound name is in scope only after the `val` binding itself. The location of the associated type annotation has no effect in this regard.

9.4 Use Declarations

use-declaration ::= (`use` *filename*)

10 Scoping Rules

11 Initial Basis