ρ

CS4120/4121/5120/5121—Spring 2023

Rho Language Specification

Cornell University

Version of April 10, 2023

Thanks to the hard work and brilliant engineering of your team, Eta has been gaining market share. However, your customers are demanding the ability to create more interesting data structures. An extended version of the language, called Rho, has been been designed. Your task is to extend the implementation of Eta with the new features, and to add some further language extension of your own design.

Rho is backward-compatible with Eta, so this language description focuses on the differences. The original Eta spec is still available.

0 Changes

- No changes yet.

1 Record definitions

An Rho program may contain declarations and definitions of *record types* in addition to function definitions. A record contains some number of named fields, each with its own type.

The following code defines a record type named `Point` and an associated creator function `createPoint`:

```
record Point {
    x,y: int
}

createPoint(x: int, y: int): Point {
    return Point(x, y)
}
```

Notice that the fields `x` and `y` can both be declared at once to have the type `int`. The same abbreviated syntax can now also be used for local and global variable declarations, though no initializer expression may be provided in that case.

Fields of a record can be accessed with the usual dot notation, e.g., given a variable `p` of type `Point`, `p.x` signifies its field `x`.

As the example shows, the name of a record type can be used as a function to construct new values of the record. This function expects to receive as many arguments as there are fields, in the order they are declared.

Records are passed by reference. For example, the following function would swap the x and y coordinates of any Point passed to it:

```
swap(p: Point) {
    t:int = p.x
    p.x = p.y
    p.y = t
}
```

Field definitions may optionally be separated by a semicolon.

2 Record declarations

All fields of a record type are visible everywhere inside its defining module (i.e., source file). To be used from a different module, the record type must be *declared* in an interface. A record type declaration looks like

a record type definition except that it may declare only a prefix of the full set of fields.

For example, we might define an interface file for the `Point` type, hiding the `x` and `y` fields from other modules:

```
// A 2D Point with integer coordinates (x,y).
record Point {}

// Create the point (x,y).
createPoint(x: int, y: int): Point

// Get the X and Y coordinates of the point
pointX(p: Point): int
pointY(p: Point): int
```

Field declarations may optionally be separated by a semicolon.

3 Modules and interfaces

The extension `.rh` is used for files containing module definitions and the extension `.ri` is used for interface files. Therefore, the statement `use my_module` appearing in a module causes the compiler to look for the interface of that module in the file `my_module.ri`.

The syntax of interfaces has also been extended so that, like modules, they can begin with "`use`" statements. If a module uses an interface, it also implicitly uses every interface that is used by that interface. It is legal for an interface to be used by a module along multiple "`use`" paths.

If a module is defined in a file `a_module.rh`, the compiler automatically looks for an interface in file `a_module.ri`, exactly as though the module had contained a statement `use a_module`. However, such an interface need not exist.

If a module that defines a record type `T` references (either explicitly or implicitly) an interface that declares type `T`, the type definition must match the fields given in the declaration. The fields and their types declared in the interface must be the same as those defined in the module, or a prefix.

A module does not need to have an interface. Further, even if there is an interface, not every type, procedure, or function in the module needs to be declared in that interface. Undeclared module components are analogous to private classes or private methods in Java because they are not visible outside their own module. However, everything declared in the interface must be defined in the module.

Globals, including record types and functions, are in scope throughout their defining module and throughout any module that uses an interface that declares them. For example, a record type can appear in its own declaration, e.g.:

```
record Node {
    value: int
    next: Node
}
```

However, global variables cannot be declared in interfaces, so they are private to their module. Different global variables in different modules are different variables even if they happen to be declared with the same name.

3.1 Null

Like C and Java, Rho incorporates Hoare's "Billion-Dollar Mistake". The special value `null` is a member of all record types and also a member of all array types. It is the default initialization value for all variables with record or array type, including global variables and array elements.

It is a run-time error to perform any operation on `null` that is specific to records or arrays. The value `null` must be implemented as a pointer to memory location 0. A reference to this memory location will cause

a page fault that will halt the program. This is an adequate implementation of the run-time checking for null values.

3.2 Other defaults

The default initialization values for `int` and `bool` are 0 and `false`, respectively.

4 New operators

The equality comparison == and the inequality comparison `!=` can be used on two value of the same record type T. These comparisons are implemented as pointer comparisons, much like in Java. However, the special value `null` can always be compared against any value with record or array type.

5 Other extensions

5.1 Assignment

To make type-checking convenient, we introduce a new judgment $\Gamma \vdash e : \tau$ lvalue, which means that e is an expression that can be assigned values of type τ (an *lvalue*). With this rule, we can rewrite the (ASSIGN) rule in a more generic form that encompasses the old rule and also the old (ARRASSIGN) rule:

$$\frac{\Gamma \vdash e_1 : \tau \text{ lvalue} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \texttt{ = } e_2 : \Gamma} \text{ (ASSIGN)}$$

The lvalues are variables, array elements, and fields:

$$\frac{\Gamma(x) = \texttt{var } \tau}{\Gamma \vdash x : \tau \text{ lvalue}} \text{ (VARLVALUE)} \qquad \frac{\Gamma \vdash e_1 : \tau\texttt{[]} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} : \tau \text{ lvalue}} \text{ (ARRLVALUE)}$$

$$\frac{\Gamma \vdash e_1 : C \quad \texttt{record } C \texttt{ \{} \ldots \texttt{ } f\texttt{:}\tau \texttt{ } \ldots \texttt{\}} \in \Gamma}{\Gamma \vdash e_1\texttt{.}f : \tau \text{ lvalue}} \text{ (FIELDLVALUE)}$$

5.2 Break statement

To make it easier to end loops, a `break` statement similar to that in C and Java has been added. As in these languages, its effect is to immediately terminate the closest lexically enclosing loop. As with the `return` statement, its type is `void`, but the `break` statement is legal only if lexically enclosed by a loop. This rule could be modeled in the type system by adding a binding β : `unit` to the context of `while` loop bodies.

6 ABI

To allow different implementations to interoperate, the ABI specifies that fields of records are laid out in the order they are declared, with each field taking up one 64-bit word. The representation of a record value is a pointer to its first field.

Top-level functions that take or return record references should encode their types into function names as follows:

1. "`r`"
2. A number giving the length of the unescaped type name.
3. The name, with underscores escaped in the usual fashion.

For example, a function with the signature `average(a: Point, b: Point): Point` would have its name encoded as `_Iaverage_r5Pointr5Pointr5Point`.