Intercept Programmer's Manual

Contributors

June 17, 2023

Introduction

With a language as extensive as `Intercept`, it can be hard to know where to start.

A program is executed from top to bottom, and in order. The entry point of the program is at the beginning of the file.

The last expression of the program is the return status code from the `main` function in the C runtime.

Let's begin.

Variables

In mathematics, a *variable* is defined as a symbol and placeholder for any mathematical object. An `Intercept` variable is a symbol (or name) and placeholder for a unique chunk of memory. Using this symbol and placeholder (the variable name), this chunk of memory can be (at least) accessed to return a value, (possibly) modified, and even more that we will get to later.

On top of this, `Intercept` variables have a type associated with them. A type determines both the size of the chunk of memory, as well as how this chunk of memory is treated.

Declaration

A variable declaration is the act of declaring a new variable within an `Intercept` program. This variable will have a unique chunk of memory associated with it. How this chunk of memory can be accessed and altered is determined by the variable's declared type. The type is declared by the programmer along with the variable symbol.

Let's take a look at a variable declaration in practice. Here is some `Intercept` source code showing some variable declarations:

```
;; In Intercept, comments start with ';;' and end at the newline.

;; Intercept Variable Declaration Syntax
;; <variable_name> ":" <variable_type> [ "=" <expression> ]

x : integer = 34
y : integer = x + 1
z : integer = x + y
```

The `:` is the *type annotation* symbol. It associates the symbol (or name) on the left hand side with a chunk of memory (at least) large enough to store the type on the right hand side. Optionally, an expression can be given that will be the initial value of the variable. The expression must result in the same type as has been declared, or a *type error* will occur. A type error indicates that a program is invalid, and the `Intercept` tooling will not move on any further with it. The above example program contains no errors of any kind.

There is also another declaration syntax with slightly different semantics:

```
;; Intercept Type-inferred Variable Declaration Syntax
;; <variable_name> "::" <expression>

x :: 34
y :: x + 1
z :: x + y
```

The above program is identical to the first, however, as you can see, it's much more concise. Because an initialising expression is required as part of this

syntax, we can use the result type of the expression as the declaration type of the variable.

Assignment

Most variables can be assigned to at any time after they have been declared using the assignment operator: `:=`.

```
;; Intercept Variable Reassignment

x : integer = 69
y : integer = x
x := 420
```

Types

NOTE: This section is likely to change in the very near future, and is outdated with incorrect information.

`Intercept` has both user-defined and built-in types. Here, they are split into two categories, *base* and *complex*, and listed with their symbol (or name) and a brief description.

Pointers and complex types are special in that they are not declared by writing the type symbol itself, but rather with special syntaxes for each. This is necessary due to the extra information that is needed from the programmer with these sorts of types.

Base Types

integer An 8 byte signed integer number.

byte A 1 byte signed integer number.

pointer The address of a chunk of memory that is at least the size of the base type.

Pointers

A pointer is a memory address; a value specifying where to find a chunk of memory at least as big as the underlying type.

To create a pointer from a variable of a valid base type, use an address-of expression.

To read the base type value from a pointer, use a dereference expression.

When declaring a type, the programmer must declare how many levels of pointer indirection are wanted. This is done using the amount of pointer declaration operators, `@`, preceding the type symbol.

```
;; Intercept Pointer Type Declaration

a : integer = 69

;; Pointer to an integer
b : @integer

c : @integer = &a
b := c

d : integer = @b
```

See `examples/pointers` in the repository for more.

Complex Types

array A contiguous chunk of memory big enough to store it's capacity amount of it's base type.

function A pointer to the beginning of a procedure that may be called and returned from.

Arrays

For an array, the programmer must declare it's base type, as well as it's capacity.

```
;; Intercept Array Type Declaration
eight_ints : integer[8]
```

An array of eight integers is a chunk of memory large enough to hold, you guessed it (Frank Stallone), eight integers. The memory address of any item in an array can be accessed, and further dereferenced to produce a value of the base type.

Functions

A function type has a return type, and a list of parameter types.

A function type's declared return type is used to know the return type of the function call expression that calls a function of function type. Basically, the function, when called, returns a value that is of the return type. A function type's declared parameter type(s), if any, are used to ensure both the proper amount of arguments, and to ensure each argument expression returns the type that the function expects.

Because a function is just another variable type, functions can be passed as arguments to other functions. In computer science, this is referred to as the downwards funarg problem, and we have solved it :P.

As for the *upwards* funarg problem, there are plans for this to be supported. The only problem at the moment is how the function type is parsed, in that the return type must be a base type. This is just a limitation of the parser, and will eventually be fixed to allow for functions to be returned from functions, as well. One reason this is possible is because `Intercept` does not allow a function to reach outside of it's own scope, other than the global scope. That is, a nested function's body may not access variables that were defined in a parent function. This allows for the parent function's stack frame to be able to be cleaned up when the function returns, and the nested function does not depend on that nested stack frame.

The programmer must declare a function's return type, as well as the types of any and all parameters. It is clear that just writing `function` is not going to cut it... This is why we have the following function type syntax:

```
;; Intercept Function Type Declaration

;; Define 'square' as a function that returns an integer,
;; and takes in one integer argument.
square : integer(n : integer)

;; Define 'plus' as a function that returns an integer,
;; and takes in two integer arguments.
plus : integer(a : integer b : integer)

;; A comma separating each parameter is optional.
sub : integer(a : integer, b : integer)
```

Attributes

Currently, there is only one attribute: *external*. It is currently only allowed on function types, and is needed to call functions that are defined outside of `Intercept`. In the future, we hope to allow any type to be external. An external type is prefixed with the `ext` keyword.

In the future, we hope to provide a *public* attribute that will allow symbols in our language to be used by other languages and such during linking.

Casting Between Types

It is not yet supported, but very soon we will work on being able to convert base types to other base types, as well as reinterpret pointers to some base types to other base types.

Here is an idea of the current syntax:

```
;; Intercept Type Casting Syntax Experimental
a : byte = [byte]69
```

We haven't yet talked about expressions, but when we do, we will learn that `69` is an expression that returns an `integer`. An integer is not a byte, and I think we can all see that this is a problem. In fact, this is a type error. However, by taking advantage of type-casting, we can convert our integer into a byte, allowing the program to stay valid with no type errors. This kind of type cast is only allowed between specific built-in types, as it requires runtime handling of the value for proper conversion.

A reinterpret type cast is more simple. It occurs between pointer types, as long as the cast type is a pointer that has a base type that is smaller or equal to in size than the expression return type. Let's see this in practice:

```
;; Intercept Reinterpret Type Casting Experimental
x : integer = 69
;; "foo" is a pointer to an integer initially assigned to the address of "x
foo : @integer = &x
bar : @byte = [@byte]foo
```

As you can see, it is valid to cast an integer pointer to a byte pointer. This is because, when the memory is read from, the integer chunk of memory is large enough to contain a byte chunk of memory. There will be no memory errors when doing this.

In the future, it would be cool to allow for casting an integer to an array of bytes. This would effectively be unpacking. It would be really, ultra cool if we could also do it the other way around, and pack an array of 8 bytes into an integer. I don't know if it would ever be useful though :p.

Expressions

An `Intercept` expression is anything that returns a value.

Numbers

The simplest expression is just a number.

```
;; Intercept Simple Integer Expressions
5
9
69
420
```

Number expressions return an `integer_literal` type, which may be cast to any other integer type. This is what allows an integer variable to be assigned to a number without a type error occurring.

Binary Infix Operators

A binary operator is a symbol that acts on two values. Usually these are referred to as the left hand side and the right hand side.

Here are the binary operators currently supported.

- $+$ Add `integer` to `integer` and return the `integer` sum.
- $-$ Subtract RHS `integer` from LHS `integer` and return the `integer` difference.
- $*$ Multiply `integer` by `integer` and return the `integer` product.
- $/$ Divide LHS `integer` by RHS `integer` and return the `integer` quotient.
- $\%$ Divide LHS `integer` by RHS `integer` and return the `integer` remainder.
- $\langle\langle$ Shift the bits of LHS `integer` by RHS `integer` to the left and return the `integer` result.
- $\rangle\rangle$ Shift the bits of LHS `integer` by RHS `integer` to the right and return the `integer` result.
- $=$ Return `integer` 1 iff `integer` and `integer` are equal; otherwise return zero.
- $\langle$ Return `integer` 1 iff LHS `integer` is less than RHS `integer`, otherwise return zero.

〉 Return `integer` 1 iff LHS `integer` is greater than RHS `integer`, otherwise return zero.

```
;; Intercept Binary Infix Operator Expression Example
34 + 35   ; 69
40 % 16   ; 8
10 < 5    ; 0
2 << 6    ; 128
```

In the future, we hope to allow for users to define their own binary operators in the program itself.

Variable Access

A variable that has been declared may be accessed using the symbol (or name) the variable was declared with.

A variable access expression returns the declared type of the variable being accessed.

The variable symbol (or name) can be thought of as a placeholder for the underlying value, much like variables in mathematical equations.

```
;; Intercept Variable Access Expression Example

;; Declaration
foo : integer
bar : integer

;; Assignment
foo := 69
bar := foo

;; Access
foo
bar
```

Lambdas

A lambda is an expression that returns a function.

```
;; Intercept Lambda Expression Example

;; A lambda that returns a function that returns an integer, and
;; doesn't take any parameters.
integer() {
  a : integer = 69
  ;; The last expression in a lambda body is the return
  ;; value.
  a
}

;; The function that the lambda returns may be immediately invoked
;; using a function call expression (parentheses).
integer(x : integer  y : integer) {
  x + y
} (34 35)

;; Assigning a variable of function type to a lambda that returns a
;; matching function type.
double : integer(n : integer) = integer(n : integer) {
  n + n
}
```

In the future, we hope to provide a function variable declaration syntax that does not include repeating the function signature for both the variable's declared type as well as the lambda expression.

Function Calls

A function variable access may be transformed into a function call using parentheses following the variable symbol. Within the parentheses, arguments are passed in order. A type error occurs when the given argument expression's result type does not match the function type declarations parameter type. The return type of a function call expression is the same as the return type of the function being called.

```
;; Intercept Function Call Expression Example

double : integer(n : integer) {
  n + n
}
answer_to_life_universe_and_everything :: double(21)

sum : integer(a : integer b : integer) {
  a + b
}
sum(34, 35)
;;  a   b
```

Dereference

The dereference operator, `@`, is unary, in that it operates on a single value. This value *must* be of a pointer type, otherwise a type error occurs.

Most of the time, a variable of pointer type is dereferenced to access the value of the type that is pointed to. Effectively, the dereference operator removes one level of pointer indirection. This kind of dereference is an expression, and returns one less level of indirection than the given pointer type. For a concrete example, an integer pointer, when dereferenced, produces an integer value.

In the specific case of a dereference on the left hand side of a variable assignment, a dereference is no longer the same type of expression. In this case, instead of returning the value, it returns the memory address of the value pointed to. This allows the assignment to write into it's left hand side result and do what the programmer expects. If this is confusing, when you see it in practice in the following example, an undertanding should hopefully fall into place.

```
;; Intercept Dereference Expression Example
a : integer = 69
b : integer

;; Declare a pointer to an integer variable.
ptr_to_a : @integer

;; Assign 'ptr_to_a' to the address of 'a'.
ptr_to_a := &a

;; Dereference 'ptr_to_a' to access the integer value pointed to.
b := @ptr_to_a

;; 'b' is now equal to the value of the integer 'ptr_to_a' pointed to.
;; Because 'ptr_to_a' pointed to 'a', 'b' is assigned the value of 'a'.
;; 'a' was last assigned to the integer '69'.
;; 'b' now equals the integer '69'.
b

;; Here is the special case, on the left hand side of an assignment.
;; A dereference is used to write to the memory pointed to by the given
;; pointer.
@ptr_to_a := 420

;; Accessing 'a' now returns the integer '420'.
;; This is due to the dereference assignment.
a
```

Address Of

The address-of operator, `&`, is used to create a pointer to a variable that has already been declared.

The address-of operator return type is the declared type of the given variable plus one level of pointer indirection.

```
;; Intercept Address Of Expression Example
a : integer
ptr : @integer
a   := 69

;; Assign 'ptr' to the address of 'a'.
ptr := &a

ptr_to_ptr : @@integer
ptr_to_ptr := &ptr
```

Array Index

An array index expression is used to access a memory address (hopefully) within an array. It returns a pointer to an item within the array. This is unlike most other languages, where the array index expression returns the value of the item itself. To get the value of the item, use a dereference expression.

```
;; Intercept Array Index Expression Example
int_array : integer[8]

;; The array index operator returns a pointer to the array base type.
first : @integer = int_array[0]

;; To access the base type value, use a dereference.
x : integer = @int_array[0]
y : integer = @int_array[1]

;; Assign new values into the array.
@a[0] := x + y
@a[1] := x + 35
```

If/Else

Unlike most languages, `Intercept`'s if/else control flow construct is an expression. This means that it returns a value.

The `else` branch is optional, but, if included, must return the same type as the `if` branch. Otherwise, static type checking would not be possible (the program would become ambiguous).

The condition of the `if` statement is tested against zero. That is, iff the condition result is zero, the `if` branch will not be taken. If one is present, the `else` branch will be taken.

```
;; Intercept If/Else Expression Example

less_than_100 : integer(n : integer) = integer(n : integer) {
  ;; Because the 'if/else' expression is the last expression
  ;; in the function, it's result is what will be returned.
  ;; The type checker ensures both branches return an 'integer'.
  if n < 100 {
    1
  } else {
    0
  }
}
```

Macros

A macro is a *rule* or *pattern* that specifies how a certain input should be mapped to a replacement output.

Defining a Simple Macro

A macro must have a minimum of three things: a name (which must be an identifier), an input list (the parameters), and an output list (the expansion).

First, let's look at the most basic macro: *empty*.

```
macro empty emits endmacro
```

Here the macro *empty* is defined as a macro that takes no inputs and emits no outputs. Usage of the *empty* macro would do nothing, as it is replaced with it's output: nothing.

Hygiene

Staying Clean in Intercept

In Intercept, macro expansions are guaranteed to be hygienic at parse time. That's right: an error will be issued if a macro expansion is unhygienic, and the program will not compile successfully. To avoid users running into these sorts of errors all the time, however, an astute Intercept programmer may write macros in such a way (i.e. using `defines`) that it is impossible for an unhygienic expansion to occur.

```
macro foo $t emits
  a :: 35
  $t + a
endmacro
```

Let's consider a use case of the macro above, where a programmer would like to call it twice in a row.

```
foo 34
foo 34
```

And after expansion:

```
a :: 35
34 + a
a :: 35
34 + a
```

Uh oh! This program no longer compiles, as a is declared twice in the same scope. So, as a macro author, how can we ensure the user can call it as many times as they want, without causing issues like this? Enter `defines`.

`defines` is an optional part of a macro definition that allows the author to declare that the macro defines a unique variable somewhere in it's output.

```
macro foo $t
defines a
emits
  a :: 35
  $t + a
endmacro
```

Now, the same usage as before:

```
foo 34
foo 34
```

And after expansion...

```
__G_xX_1_Xx_ :: 35
34 + __G_xX_1_Xx_
__G_xX_2_Xx_ :: 35
34 + __G_xX_2_Xx_
```

Woah! What happened there? Well, the Intercept compiler now knows that any usage of the symbol *a* within the expansion of the *foo* macro should actually be unique per macro invocation (as each invocation should define that variable). So, for each invocation, the symbol *a*, when encountered within the macro expansion output, is replaced with a program-wide, unique symbol. This means each invocation acts as expected, and the program is able to compile successfully.

This isn't the only problem `defines` solves; because it allows a macro author to bind a symbol in the output to a program-wide unique symbol, `defines` is basically a fancy syntax for `gensym`. This means that a macro may define a variable in a child scope without worry of it shadowing parent declarations. For example:

```
macro foo $t
defines a
emits {
  a :: 35
  $t + a
} endmacro

a :: 34
foo a
```

Expansion with `defines`:

```
a :: 34
{
  __G_xX_1_Xx_ :: 35
  a + __G_xX_1_Xx_
}
```

Expansion without `defines`:

```
a :: 34
{
  a :: 35
  a + a ;; 70?! Uh oh, that's not right!
}
```

As you can see, without generating a unique symbol, it is possible for very confusing expansions to occur. Well, it would be, if Intercept didn't check specifically for this case and emit an error in it's presence.

These are just a few ways to stay hygienic in Intercept while writing and using macros.