Function Inference

Michael Lynch

March 7, 2019

Contents

minted amsmath

1 Motivation

Function inference allows for a function to have multiple overloads for the same parameters types, but return values of different types depending on the context in which the function was used. Function inference forms an integral part of NoSyn's extention framework, allowing the user of the language to create a DSL or general purpose language within NoSyn with great capability to customize it's syntax.

1.1 Usage Example

Consider the following expression:

```
2:6
```

The user may with wish this to mean concatenating the two numbers into a list of integers. But there are also languages in which this symbol is used to denote to the start and the end of an array slice:

```
//GO CODE
a[2:6]
```

NoSyn provides the user with the ability to overload all operators, infact by default no operators are defined. Assuming that an overload function has been created for the square brackets on an array to perform array access, the user could set up the language as follows:

```
//NO SYN code
[T] operator_:_<T>(T a, T b) {
  native_formArray(a,b)
}
[T] operator_:_<T>(T x, [T] xs) {
  native_formArray(x,xs)
}

T operator_[]_<T>([T] array, Int index) {
  native_indexAccess(array, index)
}
[T] operator_[]_<T>([T] array, [Int] index) {
  native_sliceArray(array, index)
}

//Assuming 'a' is an Integer array.
[Int] slicedArray = a[2:6]
```

This system will work providing the user with the capability to use the colon symbol as both an array contatentation operator as well as an array slice operator. But this way comes with a serious flaw:

```
[Int] slicedArray = a[2:4:5:6]
```

This should be considered a syntax error. But it's not. As the operator overload for [] is expecting to see an array, `2:4:5:6` is a perfectly plausible

expression to put inside the square brackets. Of course the error could be picked up in the `native_sliceArray` expression but this would mean that the check would only be made at runtime. This is where function inference can become very useful.

1.1.1 Function Inference Example

```
//NO SYN code
[T] operator_:_<T>(T a, T b) {
  native_formArray(a,b)
}
(Int, Int) operator_:_(Int l, Int r) {
  (l,r)
}
[T] operator_:_<T>(T x, [T] xs) {
  native_formArray(x,xs)
}

T operator_[]_<T>([T] array, Int index) {
  native_indexAccess(array, index)
}
[T] operator_[]_<T>([T] array,  (Int, Int) arraySlicer) {
  native_sliceArray(array, index)
}

//Assuming 'a' is an Integer array.
[Int] slicedArray = a[2:6]
```

This new implementation allows for both element concatenation and array slicing while maintaining the ability to throw an error at compile time when `a[1:2:3]` is provided. This code still has a problem however. The operator overload that has been created returns a tuple of two ints. This is not ideal as the syntax that was specifically designed for generating an array slice would also be used in other contexts:

```
//Unwanted ability to create regular tuples
(Int, Int) vector = 20:30
//Unwanted ability to use tuples within the array access
[Int] slicedArray = a[(2,6)]
```

In order to prevent this usage of the colon operator, a closed alias can be used:

```
alias closed ArraySlicer = (Int, Int)

ArraySlicer operator_:_(Int a, Int b) {
  (l,r)
}

operator_[]_<T>([T] array, ArraySlicer arraySlicer)_{
  native_sliceArray(array, arraySlicer)
}
```

Using this implementation. The colon operator overload function will only be infered if the context in which it is used is specifically of the type `ArraySlicer` and not simply `(Int, Int)`.

2 Algorithm for Function Inference

Function inference uses a type inference algorithm to work out which function overload to use. Where as type inference is often used to save the programmer time by not requiring the user to specify what the type of a variable or function is, function inference expects a certain amount of information about the context it is being used in. As such, **NoSyn** does not allow the user to specify a variable without also specifying the type of that variable. Functions similarly must indicate what the return type is, although template types are still valid. This constraint employed in using function inference is used to reduce the ambiguity which can arise using such a type system. Later I will explore ways in which type inference of variables may be possible alongside function inference, and why such a feature may not be wanted.

2.1 Formal Definition of Function Inference Algorithm

2.1.1 Function Definitions Used Within Algorithm Formula

$\Lambda(r, p)[y] \Rightarrow k$
$\Lambda r[y] \Rightarrow k$

- where r is the set of all possible return types for function call y
- where p is the list of sets of all possible return types for the parameters of y
- k si the set of all possible function overloads for function call

$\Omega(r, p)[y] \Rightarrow k$
$\Omega(r)[y] \Rightarrow k$

- Where r is the set of all possible return types for a function call
- Where p is the list of sets of possible parameter types for a function call
- Where k is the set of all possible function overloads given r and p for function call y

$\Theta z \Rightarrow n$

- Where z is a set of possible function overloads
- n is the list of sets of all possible return types for the parameters of function calls from the given overloads

$\Phi x \Rightarrow m$

- where z is a set of possible function overloads
- m is the set of all possible return types for those function overloads

$y^\dagger$

- where y is a function call
- $y^\dagger$ is a list of function calls for the parameters on the function call y

All parameters can be assumed to be function calls as literals can be expressed as function calls to functions with single overloads and no parameters

2.1.2 Algorithm Formula

The algorithm for function inference can be written as:

$$\begin{aligned} \Lambda x[y] := \\ let\ p := [\forall(\alpha, \beta).\Theta\Lambda\alpha[\beta] | zip(\Theta(\Omega x[y]), y^\dagger)]\ in \\ \Lambda(\Phi(\Omega(x, p)[y]), p)[y] \end{aligned}$$

The function $\Lambda x[y]$ calls recursively until $\Omega(x, p)[y]$ reduces to only a single possible function overload. If $\Omega(x, p)[y]$ never reduces to a single function overload, the function call is ambigiuous and a compile error should occur.

2.2 Example Inferences

2.2.1 Context Deduction

Function inference works on the basis of deducing the context in which a function is being used. All function calls are expressions and can be built up into larger expressions.

- All expressions have a single type
- Expressions can be used as a statement if they have the type `Nothing`
- literals have a clear concrete type

Using these rules we can deduce that given the following statement:

```
foo(10)
```

- The type of the expression `foo(10)` must be `Nothing` as it is being used as a statement
- The function overload of `foo` is `Int->Nothing` as the literal `10` has the concrete type of `Int`

As the *NoSyn* language, unlike similar languages like *C*, allows for functions with the same name and parameter types to have multiple different return types, expressions already raise an issue of ambiguity. *C* and *Java* would in this situation go for the function overload for `foo` which took a single integer as a parameter and then ignore the return type. This is not possible in *NoSyn* due to the potential for there to be multiple overloads with the same parameter types causing ambiguity. Instead there is a special datatype which a function can return if the programmer wants to use a call to the function at the statement level. This is the `Nothing` datatype, which as it's name suggests, does not return anything. An expression of the type `Nothing` is never a subexpression of another expression as `Nothing`. This is because `Nothing` does not have any value and as such cannot be passed into any other function. With this knowledge, we always know that the base type of any expression within the language is of type `Nothing`, and all subexpressions in that expression are of some non `Nothing` type.

2.2.2 A slightly less simple program

```
//foo_IntNothing
Nothing foo(Int a) {..}
```

```
//foo_IntInt
Int foo(Int a) {..}
//bar_Int
Int bar() {..}
//bar_Float
Float bar() {..}

foo(foo(bar())) //Expression A
```

Expression A is an example of where function inference is required to find the correct function to be used. If you take the subexpressions of expression *A* out of context, the functions they refer to cannot be known:

- `bar()` may refer to `bar_Int` or `bar_Float`
- `foo(bar())` may refer to `foo_IntNothing` or `foo_IntInt`

In order to deduce the type of each subexpression, we must work from the information that we know concretely. The base expression `foo(foo(bar())` must return `Nothing` as it is being used as statement. From this we can gather all the function overloads for foo which return `Nothing`. In this simple program there is only one function which this could be, `foo_IntNothing`. Given this information, we can now deduce that the subexpression `foo(bar())` must be of type `Int` if it is to satisfy the base expression. Again, as a simple program, there is in this case only one function which `foo` could be refering to: `foo_IntInt`. This then gives us the knowledge to work out what our final subexpression refers to. There is one function overload for `bar` which returns an `Int` which is `bar_Int`. This completes the deduction of all functions in the expression giving us:

```
foo_IntNothing(foo_IntInt(bar_Int()))
```

2.2.3 Horizontal Inference

With the previous example, the correct function overloads could be infered by working in a top down fashion from the parent expression `foo(foo(bar()))` down to the leaf subexpression `bar()`. This can be refered to as vertical inference in the sense that be looking at the context an expression or it's subexpressions it is possible to infer the type of the expression. Horizontal Inference means that the type of a subexpression on the same level as the current one has an effect on the type which this subexpression could be. Such inference is achieved by working up and down the expression tree gradually

eliminating the possible types of expressions until all are resolved down to a single type.

```
Nothing foo(Int a, Double a) {..} //foo_IntDoubleNothing
Nothing foo(Int a, Char a) {..} //foo_IntCharNothing
Nothing foo(Double, Int a) {..} //foo_DoubleIntNothing
Int bar() {..} //bar_Int
Char bar() {..} //bar_Char
Int cello() {..} //cello_Int
Double cello() {..} //cello_Double

foo(bar(), cello()) //Expression B
```

2.2.4 Applying the Function Inference Algorithm to Expression B

As with before, each expression within expression B cannot on its own have it's function infered. As with before. As the main expression B is being used as a statement, the type can be infered to be `Nothing`. As such, the set of possible `foo` functions expression B could refer to is:

`{Nothing}{...}foo -> {foo_IntDoubleNothing, foo_IntCharNothing, foo_DoubleIntNothing}` (`{Nothing}(?,?)foo` means to find all the possible function overloads for `foo` with return type `Nothing` and 2 parameters of any type)

From this list of possible functions, a list of possible parameter types can be infered:

- Parameter 1: `{Int, Double}`
- Parameter 2: `{Double, Char, Int}`

With these sets of parameter types, these can be applied to the parameter expressions `bar()` and `cello()`:

- `{Int, Double}bar -> {Int}`
- `{Double, Char, Int}cello -> {Double, Int}`

With these reduced sets of parameter types this can be then applied again to the `foo` function overloads to see if the number of possible overloads can be reduced: `{Nothing}({Int}, {Double, Int})foo -> {foo_IntDoubleNothing}`

This finds the only possible function overload that `foo` can be refering to as `foo_IntDoubleNothing` allowing for the whole expression to be infered as:

```
foo_IntDoubleNothing(bar_Int(), cello_Double())
```

This can be considered horizontal inference as the type of `bar` has a direct effect on the type of `cello`. Had the possible function overloads for `bar` have been:

```
Double bar() {..}
Char bar() {..}
```

Then the expression would have evaluated as:

```
foo_DoubleIntNothing(bar_Double(), cello_Int())
```