Slot Lambda Calculus

by Sven Nilsen, 2017

In this paper I introduce a modified lambda calculus that allows functional programming languages to construct new programming languages very easily. I also derive the rules for classifying edge cases for the generated syntax, which makes it easier for soundness analysis. This is a valuable tool when studying path semantics.

(_ _)(?)(_ _)("+")(\) = (? _)(_ _)("+")(\) = (? (_ _))("+")(\) = (? ("+" _))(\) = (_ + _)

(_ + _)(1)(2) = (1 + _)(2) = (1 + 2) = 3

Slot lambdas differs from normal lambdas in four ways:

1. The application rule is changed such that when you pass a slot lambda to a slot lambda, you get a new slot lambda that you must "fill up" with the missing arguments before continuing.
2. New special terms are added to use slot lambdas to extend its own syntax.
3. Parsing rules and serialization to code can be derived.
4. One can pattern match on slot lambdas to extract expressions.

Intuitively, the slot lambda of addition can be written:

_ + _

If we pass `1` to `_ + _`, we get a new slot lambda, similar to a normal lambda:

(\x => \y => x + y)(1) == (\y => 1 + y)	Normal lambda
(_ + _)(1) == (1 + _)	Slot lambda

One difference between a normal lambda and a slot lambda is that you can pass a slot lambda to obtain a new slot lambda with an intuitive result:

(1 + _)(_ + _) = (1 + (_ + _))

With a normal lambda application rule, it would try adding `1` with the lambda argument, which is incorrect. This leads to an error instead of returning a new lambda that takes more arguments. Slot lambdas do not have this problem.

You can not use slot lambdas for higher order functional programming the same way as with normal lambdas. However, it is possible to use normal evaluation rules for normal lambdas.

A cool thing about slot lambdas is that you can use them to create new syntax. A symbol `?` is used to fill slots temporarily, and `\` is used to erase all occurrences of `?` and turn all strings into syntax.

This means we can start with a very tiny language for slot lambdas and construct new languages within it.

(_ _)	Pair
?	Placeholder
\	Transform to syntax
“abc”	String
1, 2, …	Number
x, y, z	Variable

For example, to create the syntax for addition `_ + _`, you write the following:

(_ _)(?) = (? _)	Insert `?` because we need `+` after an empty slot
(? _)(_ _) = (? (_ _))	Add a new pair to not run out of empty slots
(? _ _)(“+”) = (? (“+” _))	Insert a string “+” that will be the operator symbol
(? (“+” _))(\) = (_ (+ _))	Erase `?` and turn “+” into syntax

By putting all these operations together, you can define `_ + _` in a short program:

(_ _)(?)(_ _)(“+”)(\) = (_ + _)

To make it easier to read, one can write an equation that tells how to interpret the syntax without the parentheses:

(_ (+ _)) = (_ + _)

Pattern matching with slot lambdas makes it possible to work with the syntax without parentheses and not worry about how the underlying tree is structured:

pattern((a + b), (_ + _)) = [a, b]

One problem that language designers face, is checking that all syntax rules have an interpretation. This is called “well formed expressions”. The rules for well formed expressions in slot lambda calculus must be decided by the designer, but to do this one needs to first classify all the edge cases.

For example, to create a small language with addition `_ + _` as the only operator, one must check that there are no unintended ways of constructing code that do not have an interpretation by the compiler. We use the letter `a` to annotate an expression that is not a slot lambda. All constructions we need to check are the following:

(_ + _)(a) = (a + _)	left value
(_ + _)(_ + _) = ((_ + _) + _)	left sub addition
(a + _)(a) = (a + a)	ready for evaluation
(a + _)(_ + _) = (a + (_ + _))	right sub addition

Whenever we get a new slot lambda with an empty slot that is not inside an inner slot lambda, then this new slot lambda must also be checked, for example `(a + _)`.

If the syntax is not context sensitive, there is no need to check the new slot lambdas where the next empty slot ends up in an inner slot lambda. This is because all inner slot lambdas will be checked by another rule, for example in the case `((_ + _) + _)` the inner slot lambda `(_ + _)` is already checked for well formed-ness. If an error happens, then the parsing/evaluation stops and reports the error. The slot lambdas `((a + _) + _)`, `(((_ + _) + _) + _)` etc. can always be constructed and interpreted by the same rules.

A language is well formed when all expressions have a decided rule.

There is no need to check slot lambdas that contain variables for right arguments, because:

1. … either it has no empty slots, e.g. `(a + a)` and has an interpretation
2. … or it can be constructed using syntax, e.g. `(a + _) = (_ + _)(a)`,
   in which case any slot lambda that takes `(_ + _)` as right argument
   will on next argument put it in an empty slot in the inner slot lambda.

Therefore:

1. When you follow the rules above that gives an error when syntax or interpretation is invalid,
2. No matter how you construct the syntax for the new language, it is well formed.

If you are still in doubt about some cases, you can decompose and check which rule handles it:

Case	Decompose		Check rule
(a + _)(a + _)	(a + _)(_ + _)(a)	=>	handled by the `(_ + _)(a)` rule
((_ + _) + _)(a)	(_ + _)(_ + _)(a)	=>	handled by the `(_ + _)(a)` rule
(a + (_ + _))(a)	(a + _)(_ + _)(a)	=>	handled by the `(_ + _)(a)` rule