# Ambiguously-Typed Lambda Calculus

Athan Clark

**Abstract**

Here, we present the ambiguously-typed lambda calculus - a size-dependent type system measuring the *shape* of terms, based on their context, and an additional *substitution system*, facilitating the merge and sort of multiple terms' parameters.

## 1 Motivation

The Simply-Typed Lambda Calculus follows from the untyped lambda calculus in that there is structural assignment to parameters, and each "step" of arity is mechanically separated with $\rightarrow$. Values are given type labels, and arguments' types are checked one-for-one to the specification signature. Higher order function application, the true nature of lambda calculus, is retained through parameter specification (or type signature) nesting. The grammars are structured as follows:

| Untyped Lambda Calculus | Simply-Typed Lambda Calculus |
|---|---|
| | $\tau ::= \tau \rightarrow \tau \mid T \quad \text{where} \quad T \in B$ |
| $e = x$ | $e = x : \tau$ |
| $\quad \lambda x.e$ | $\quad \lambda(x : \tau).e$ |
| $\quad e \;\; e$ | $\quad e \;\; e$ |
| | $\quad c$ |

$c$ is a "term constant", such that $c$ is an inhabitant of a type $T$ included in our working set $B$.

The untyped lambda calculus gives us a foundation to base all others off of - it is the minimum embodyment of higher-order function application and abstraction. But, there is no beginning, and no end; it suffices only to provide action, and not results. This is what the simply-typed lambda calculus fills - it provides an encoding of the finite "end" of an expression in it's type, by utilizing $\rightarrow$ for each step.
The simply-typed lambda calculus makes a critical decision - it gives up infinite arity for the sake of traction and decidable termination. We present the ambiguously-typing scheme to give back our infinite arity, at the cost of detailed knowledge.

# 2 Overview

Our system encodes arity in the space of variables quantified over natural numbers, and constrained based on requirements induced by application and abstraction context. This is a size-dependent type system variant, similar to Cryptol. Indeed, our "size" of terms is ambiguous - it gives us no insight to how parameters are resolved. We additionally include a *parameter resulution system* - a method for unifying substitutions. We later shoe-horn a pseudo-monoid instance to our system, with the *union* of lambdas as our monoidal append.
Our type system also has decidable and total type inference; the size-dependent system initially assumes all terms to be polymorphic in arity, then, depending on how terms are used, minimum bounds are enforced in our sizes based on natural number literals.

## 2.1 Brief Example

$$x : \forall a \in \mathbb{N}. \Rightarrow \qquad a \tag{1}$$
$$f : \forall b \in \mathbb{N}. \Rightarrow \qquad b \tag{2}$$
$$f\,x : \forall a \in \mathbb{N}, b \in \mathbb{N}.\{a \geq 1\} \Rightarrow \qquad (a-1)+b \tag{3}$$
$$\tag{4}$$

In our first examples 1 and 2, their sizes are purely polymorphic *because* there is no context telling us how the expression should behave. In 3, we can see some interesting ideas: because $f$ was applied to $x$, we now have a constraint bound to it's type variable[1]. Also, because $x$ consumed one parameter in $a$, we must decrement it. Lastly, we take the left-over parameters in $x$ and $a-1$ and combine them; in our (commutative) sized interpretation, this is simply addition[2].

[1] A degenerate consequence of our structureless arity specification is that a type variable's reference to it's term must be syntactically in-order - $\forall\, a\, b$ over $x\, y$ will match $x$ with $a$, and $y$ with $b$.

[2] This neglects the order that the parameters get combined intentionally.

Note that I didn't include the type of a lambda. Please be patient; we will find that a function's size depends on it's body.

## 2.2 Grandiose Hand-Wave

Here is our grammar:

$$
\begin{aligned}
e = \; & x && \text{term} && (5)\\
& \lambda x.e && \text{abstraction} && (6)\\
& \lceil e\, e && \text{inner application} && (7)\\
& e \lceil e && \text{outer application} && (8)\\
& e \diamond\rfloor e && \text{append} && (9)\\
& e \lfloor\diamond e && \text{contra} - \text{append} && (10)\\
& l && \text{literal} && (11)
\end{aligned}
$$

The first four elements of our grammar are inherited from our tradtional untyped lambda calculus, with two different application styles to handle how parameters are combined - we stick with simple precedence in this draft [3] , such that $x \lceil y$ will precede $y$'s parameters over $x$'s, and vise-versa for $\lceil x\, y$.

The last three exist for our free monoid - the normal append takes it's left-most argument as most precedent, while contra-append is convenient for short-circuiting with rightward precedence. Literals are not necessary for the soundness of our system, but they will be for terminating execution - $l : 0$.

### 2.2.1 Operator Type Signatures

To give a feel for how the system works, it is important to give a description of the operators we use:

$$
\begin{aligned}
\lceil f\, x \; & : \forall ab \in \mathbb{N}.\{a \geq 1\} \Rightarrow && (a-1)+b && (12)\\
f \lceil x \; & : \forall ab \in \mathbb{N}.\{a \geq 1\} \Rightarrow && (a-1)+b && (13)\\
x \diamond\rfloor y \; & : \forall ab \in \mathbb{N}. \Rightarrow && a+b && (14)\\
x \lfloor\diamond y \; & : \forall ab \in \mathbb{N}. \Rightarrow && a+b && (15)
\end{aligned}
$$

Our monoid does not apply or reduce our parameter size, while application will. Notice that the size is commutative in our parameter stacks - even though the parameter stack in 12 and 13 are opposite, their size is the same.

[3] We could, in theory, make any coinductive zipper facilitate parameter resolution.

### 2.2.2 Elementary Term Type Signatures

For verbosity, we show the most simple terms and their types. In $\lambda$text, a literal is a Haskell `String`:

$$x : \forall a \in \mathbb{N}. \Rightarrow \quad a \tag{16}$$

$$\texttt{"foo"} : \quad 0 \tag{17}$$

$$\tag{18}$$

## 2.3 Abstraction Type Signature

In the simply-typed lambda calculus, we can't find the entire type signature of a lambda term, based solely on the lambda itself - we can see that the parameter would prepend a $a \to \ldots$, but $\ldots$ depends entirely on the body $e$ of $\lambda p.e$.
Our system here is similar, but a bit simpler:

$$p : \quad \forall a. \Rightarrow \quad a \tag{19}$$

$$e : \quad \forall b. \Rightarrow \quad b \tag{20}$$

$$\lambda p.e : if(p \in ftv(e)) \quad then\, \forall ab \in \mathbb{N}. \Rightarrow \quad (b+1)+a \tag{21}$$

$$else\, \forall ab \in \mathbb{N}. \Rightarrow \quad b+1 \tag{22}$$

In this instance, our type variables for 22 *lenses* into our lambda for brevity; our actual implementation will construct our types bottom-up, so one should not take this voodoo seriously.
However, our type signature is fairly straigt forward - if $p$ occours in $e$, then it's arity must obviously be included in our lambda (but only once - even multiple occourances of $p$ in $e$ will still be delegated from $p$'s unique parameter listing) - if it's not in our function body, then there will be no delegation. This is how we handle constant functions:

$$\lambda c.\lambda \epsilon.c : \forall ab \in \mathbb{N} \Rightarrow \quad a+1 \tag{23}$$

Note that we don't type-variable match against the lambda body because it's already captured with $a$. Absurd hand wave, I know. This will be fine going bottom-up with our substitution model.

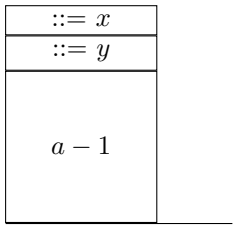
# 3 Substitution Mapping

We must have some tractable method for this to be legible and actually useful, which we attempt to present here. First and foremost is the notion of

a *substitution mapping* - an idea we can use when modeling lambdas and reduction.

We are going to sidestep everything we've worked on now to give a different perspective of how to model lambdas, via substitution.

### 3.1 Notation




This diagram can be seen as a manual abstraction in our system - we assign countable parameters at the top of our stack to names we use in our substitution body, and hand-over the rest to those bound terms. A substitution mapping has scope to only the parameters declared, both in the body and fall-over parameters.
In a substitution body, we have access to a restricted set of operators - monoidal append, term application, and literals. Free variables should not exist[4], and the operators we use in the body are the plain-jane ones we know and love, because precedence is already handled in our stack system.

### 3.2 Translation

We assume that substitution maps are the values referenced by ATLC expressions. A substitution map must be constructively finite, but referencing the "rest" (for learning purposes) is done through quantification.
Literals are 0-ary substitutions, ATLC monoidal append merges respective parameters and homomorphically settles natural append in the substitution body. Substitution application has a similar flavour; we consume our bound variable in the applyee and literally substitute the applied term's substitution body for the variable. Abstraction may either bind a free variable (when evaluating our expression), or induce a shadow on a previous binding[5]

## A Appendix Heading

[4] This will be unavoidable when we translate expressions to substitution maps.
[5] *Previous*, because we are going "bottom-up".