PizzaLang: A Language for Hungry Developers

Ronik Bhaskar and Dawson Ren

December 29, 2022

1 Introduction

It's 11pm. You've been programming all day, and you forgot to eat dinner. With the project deadline approaching, it's not like you had time. You can't keep working on an empty stomach, but you can't step away from the keyboard either.

Pull yourself together. You're a programmer. You're a professional problem solver. You can get some food and keep writing code at the same time. All you need are the right tools.

PizzaLang aims to provide the necessary tools to hungry programmers. The language has one main purpose: constructing pizzas. Using clean structure, straightforward typing, and simple grammar, *PizzaLang* allows users to construct terms suitable for any pizza-related project, including ordering pizza.

2 Term Grammar

The term grammar of *PizzaLang* is as follows:

t	::= bp	*base pizza*
	| left	*left half of pizza*
	| right	*right half of pizza*
	| all	*whole pizza*
	| x	*variable name*
	| topping x	*pizza topping*
	| add t t t	*add topping*
	| remove t t	*remove topping*

This allows for the construction of the following values. All values are specified by v, following the style of Types and Programming Languages [1]. Pizza values are denoted by pv, topping values are denoted by tv, and location values are denoted by lv.

v	::= pv	*pizza values*
	| tv	*topping values*
	| lv	*location values*

tv ::= topping x

lv ::= left
| right
| all

pv ::= bp
| add pv tv lv

Finally, this language has three types: Pizza, Topping, and Location.

$$\mathcal{T} ::= \text{Pizza} \mid \text{Topping} \mid \text{Location}$$

3 Evaluation Rules

The add and remove terms in *PizzaLang* function similarly to successor and predecessor respectively in Peano Arithmetic. The base pizza functions like zero. The predecessor of zero is still zero, so removing a topping from a plain pizza still results in a plain pizza. Wrapping add terms around a base pizza creates a larger pizza.

Unlike Peano Arithmetic, not all adds and removes are created equal. A remove around an add will only cause the two to cancel if they describe the same topping. Logically, removing mushrooms shouldn't also cause the green peppers to be removed from your pizza. Since adds can be nested, the remove term can descend through the adds, searching for the correct topping to remove.

Computation Rules

$$\frac{v_2 = v_4}{\text{remove (add } v_1\ v_2\ v_3)\ v_4 \to v_1}$$

$$\frac{v_2 \neq v_4}{\text{remove (add } v_1\ v_2\ v_3)\ v_4 \to \text{add (remove } v_1\ v_4)\ v_2\ v_3}$$

$$\frac{}{\text{remove bp } t_2 \to \text{bp}}$$

Congruence Rules

$$\frac{t_1 \to t_1'}{\text{add } t_1\ t_2\ t_3 \to \text{add } t_1'\ t_2\ t_3}$$

$$\frac{t_2 \to t_2'}{\text{add } v_1\ t_2\ t_3 \to \text{add } v_1\ t_2'\ t_3}$$

$$\frac{t_3 \to t_3'}{\text{add } v_1\ v_2\ t_3 \to \text{add } v_1\ v_2\ t_3'}$$

$$\frac{t_1 \to t_1'}{\text{remove } t_1\ t_2 \to \text{remove } t_1'\ t_2}$$

$$\frac{t_2 \to t_2'}{\text{remove } v_1\ t_2 \to \text{remove } v_1\ t_2'}$$

4 Type Rules

Since *PizzaLang* does not utilize variables beyond constructing toppings, and there is no way to evaluate or eliminate a topping, *PizzaLang* does not require a context Γ to store the types of variables. Also, depending on the implementation of the interpreter/compiler, "mushrooms" is a well-typed program.

$$\frac{}{\text{bp} : \text{Pizza}}$$

$$\frac{}{\text{left} : \text{Location}}$$

$$\frac{}{\text{right} : \text{Location}}$$

$$\frac{}{\text{all} : \text{Location}}$$

$$\frac{}{\text{topping } x : \text{Topping}}$$

$$\frac{t_1 : \text{Pizza} \qquad t_2 : \text{Topping} \qquad t_3 : \text{Location}}{\text{add } t_1\ t_2\ t_3 : \text{Pizza}}$$

$$\frac{t_1 : \text{Pizza} \qquad t_2 : \text{Topping}}{\text{remove } t_1\ t_2 : \text{Pizza}}$$

Given our type system, you may have noticed that the second, third, and fifth congruence rules are redundant, since terms of type Location or Topping are already values. These congruence rules future-proof the language for new terms of type Location or Topping that step to a value in one or more steps.

5 Type Soundness

We claim that if a term in our language is well-typed, then it steps to a value in zero or more steps. We do not claim that this proof is perfect, but it should be on the right track.

Proof. Take any term t in *PizzaLang*. Let t be well-typed. We will consider terms of type Topping and Location first. If t : Topping, then t = topping x for some variable x. Thus, t is a value, so it steps to a value in zero steps. t : Location, then t = left or t = right or t = all. In every case, t is a value, so it steps to a value in zero steps.

Consider the case in which t : Pizza. If t = bp, then t is a value, so t steps to a value in zero steps. If not, then t can be expressed either as add $t_1\ t_2\ t_3$ or remove $t_1\ t_2$. In both cases, t_1 : Pizza, so t_1 can be expressed either as an add or remove term or as bp. Since we only consider finite programs, this nesting must terminate.

At the bottom, we either have add $bp\ t_2\ t_3$ or remove $bp\ t_2$. If we have remove $bp\ t_2$, then this takes 1 step to bp, which is a value. By the congruence rules, this will be evaluated first, which means any remove terms at the bottom of the nesting will be removed first, regardless of the terms wrapped around it. If we have add $bp\ t_2\ t_3$, then t_2 : Topping and t_3 : Location, so this is a value. As such, we need only consider add terms wrapped around a bp.

Let t : Pizza be a well-typed term of k adds wrapped around a bp. If we write t' = add $t\ t_2\ t_3$, where t' : Pizza is well-typed, then t_2 : Topping and t_3 : Location, and t' is a value, so t' takes zero steps to being a value.

We want to show that for all $k \in \mathbb{N}$, if t : Pizza is well-typed and constructed with k add terms wrapped around a bp, then t' = remove $t\ t_r$ where t' : Pizza takes at most $k+1$ steps to a value. The computation rules establish a base case. Assume that for all $k \leq n$, the statement is true. Let t = add $t_1\ t_2\ t_3$. t_2, t_3, and t_r must be values as established earlier. If $t_2 = t_r$, then t' takes one step to t_1, which is a value. If not, then t' takes one step to add (remove $t_1\ t_r$) $t_2\ t_3$. By the inductive hypothesis and the first congruence rule, the inside takes at most k steps to a value. Thus, t' takes at most $k+1$ steps to a value.

Therefore, for all well-typed terms t in *PizzaLang*, they take zero or more steps to a value. □

6 Syntax

The initial interpreter, written in Typed Racket, mimics a pizza order with its syntax. It only specifies three different toppings, and they must be spelled as described. Furthermore, all programs must begin with "i'd like uh..." and end with "don't forget to bake it". We feel the first syntax requirement adds to the programming experience, mimicking common speech patterns for ordering food. The second requirement is entirely to annoy programmers who forget to add it.

t	::= "pizza"	*base pizza*
	| "with" t t	*topping and location*
	| "on the" t	*specifies location*
	| "and" t t	*topping and location*
	| "and" t	*specifies topping*
	| "hold the" t	*remove topping*
	| "actually"	
	| "left half"	*location*
	| "right half"	*location*
	| "whole thing"	*location*
	| "mushrooms"	*topping*
	| "onions"	*topping*
	| "green peppers"	*topping*

Here is an example program:

```
i'd like uh...
pizza with mushrooms on the left half
and green peppers on the right half
and onions on the whole thing
actually
hold the onions and mushrooms
don't forget to bake it
```

The program evaluates to the following term:

```
and bp (topping "green peppers") right
```

7 Further Applications

The most important values in the program are those of type Pizza. They can be read as a series of instructions for which toppings to add to a base pizza, in what order, and where on the pizza. To allow for further computations, an interpreter could use the sequence of toppings as instructions after complete evaluation. For example, you may read the location of the topping as a left/right/stay move on a Turing tape, and the string may tell you how to manipulate that square. Alternatively, you could feed the instructions to a program that draws pizzas.

Beyond computation, *PizzaLang* creates a context-free grammar for ordering pizza. Pizza delivery APIs have existed for years. Individual users can define what a base pizza means to them, and most pizza-delivery restaurants take orders as sequences of toppings to put on a pizza. The instructions may need to be adapted to fit the nuance of each pizza restaurant, but the logic is still clear.

Programmers of the world, open your editors. Let your stomachs be your guide. Write some code, and order some pizza.

References

[1] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Mass., 2002.

Special thanks to Professor Adam Shaw and Professor Vincent St-Amour.