General Function Composition

by Sven Nilsen, 2022

In this paper I present a generally computational model of function composition that formalizes, in lambda calculus, the intuition that language designers use for binary operators and parallel composition, without assuming the existence of tuples or parallel tuples.

Most people think about function composition[1] in the following way:

$$comp : (A \to B) \to (B \to C) \to (A \to C)$$

The problem is that this is just a part of what function composition means in many languages.
To talk about what languages do precisely, it needs to be formalized in a language that is universal for computation without assuming particular choices of language design.

General Function Composition is a formal model that uses Simply Typed Lambda Calculus[2]:

$$\begin{aligned}
&\because \quad comp := \backslash(f : A \to B) = \backslash(g : B \to C) = \backslash(x : A) = g(f(x)) \\
&\because \quad id := \backslash(x : A) = x \\
&\because \quad vop := \backslash(n : \mathbb{N}) = \backslash(f : A_0 \to A_1 \to \ldots \to A_{n-1} \to B) = \backslash(g : B \to C) = \\
&\qquad\qquad \backslash(x_0 : A_0) = \backslash(x_1 : A_1) = \ldots = \backslash(x : A_{n-1}) = g(f(x_0)(x_1)\ldots(x_{n-1})) \\
&\because \quad bop <=> vop(2) \\
&\because \quad vpar := \backslash(n : \mathbb{N}) = \backslash(g_0 : A_0 \to B_0) = \backslash(g_1 : A_1 \to B_1) = \ldots = \backslash(g_{n-1} : A_{n-1} \to B_{n-1}) = \\
&\qquad\qquad \backslash(f : B_0 \to B_1 \to \ldots \to B_{n-1} \to C) = \\
&\qquad\qquad \backslash(x_0 : A_0) = \backslash(x_1 : A_1) = \ldots = \backslash(x_{n-1} : A_{n-1}) = f(g_0(x_0))(g_1(x_1))\ldots(g_{n-1}(x_{n-1})) \\
&\because \quad bpar <=> vpar(2) \\
&\because \quad op(0) <=> id \\
&\because \quad op(n, f_0, f_1, \ldots, f_{n-1}) <=> f_{n-1} \cdot \ldots \cdot f_1 \cdot f_0 \\
&\because \quad par(k, n, f_{00}, f_{10}, \ldots, f_{k-1\,0}, \ldots, f_{0\,n-1\,0}, f_{1\,n-1}, \ldots, f_{k-1\,n-1}) <=> \\
&\qquad vpar(k, op(n, f_{00}, f_{01}, \ldots, f_{0\,n-1}), op(n, f_{10}, f_{11}, \ldots, f_{1\,n-1}), \ldots, op(n, f_{k-1\,0}, f_{k-1\,1}, \ldots, f_{k-1\,n-1}))
\end{aligned}$$

$$\begin{aligned}
&\therefore \quad comp : (A \to B) \to (B \to C) \to (A \to C) \\
&\therefore \quad id : A \to A \\
&\therefore \quad bop : (A_0 \to A_1 \to B) \to (B \to C) \to (A_0 \to A_1 \to C) \\
&\therefore \quad bpar : (A_0 \to B_0) \to (A_1 \to B_1) \to (B_0 \to B_1 \to C) \to (A_0 \to A_1 \to C)
\end{aligned}$$

I chose Simply Typed Lambda Calculus for this purpose, since it is Turing complete[3] and does not assume existence of tuples[4] e.g. `(a, b)` or parallel tuples e.g. `(f × g)` (Jeremy Gibbons' "product bifunctors"[5]). Category Theory[6] obscures that `h · (f × g)` is not well defined, due to currying[7]. However, simply typed lambda calculus is not expressive enough for all my needs, so I will settle with a mere translation. This approach allows me to use natural numbers[8] and variadic functions[9].

For example, normal path[10] notation can be translated the following way:

$$\begin{aligned}
&f[g_0 \to g_1] <=> op(3)(inv(g_1))(f)(g_0) \\
&f[g_0 \times g_1 \to g_2] <=> bop(bpar(inv(g_0))(inv(g_1))(f))(g_2) \\
&f[g_0 \times g_1 \times \ldots \times g_{n-1} \to g_n] <=> vop(vpar(n)(inv(g_0))(inv(g_1))\ldots(inv(g_{n-1}))(f))(g_n)
\end{aligned}$$

Where `inv` is the imaginary inverse[11].

Here is an example that derives the formula of a symmetric unary normal path:

$f[g_0 \times g_1 \times \ldots \times g_{n-1} \rightarrow g_n] <=> vop(vpar(n)(inv(g_0))(inv(g_1))\ldots(inv(g_{n-1}))(f))(g_n)$
$f[g \rightarrow g] <=> vop(vpar(1)(inv(g))(f))(g)$
$f[g] <=> g \,.\, f \,.\, inv(g)$

A parallel tuple can be thought of as the following:

$\exists\ n\ \{\ (f_0 \times f_1) <=> par(2, n, f_0, f_1)\ \}$

While parallel tuples are not fully expressible in this language, they can be translated when including the full context of the composed function.

The definition of `bop` and `bpar` can be derived:

$bop := \backslash(f : A_0 \rightarrow A_1 \rightarrow B) = \backslash(g : B \rightarrow C) = \backslash(x_0 : A_0) = \backslash(x_1 : A_1) = g(f(x_0)(x_1))$

$bpar := \backslash(g_0 : A_0 \rightarrow B_0) = \backslash(g_1 : A_1 \rightarrow B_1) = \backslash(f : B_0 \rightarrow B_1 \rightarrow C) =$
$\backslash(x_0 : A_0) = \backslash(x_1 : A_1) = f(g_0(x_0))(g_1(x_1))$

One benefit of this model is that it can be used to reason about program termination[12].

For example, imagine that one is simulating some physical system:

```
fn simulate(x : physical_state, stop_seconds : f64, dt : f64) → physical_state {
    for i (stop_seconds / dt) { x = update(x, dt) }
    return x
}
```

This can be thought of as a program composed of many sequential operations:

(simulate stop_seconds, dt) <=> op(stop_seconds / dt)((update dt); stop_seconds / dt)

This program terminates when `stop_seconds / dt` terminates and `(update dt)` is total.

The notation `(f y)` is a shorthand for `\(x) = f(x, y)`[13].
The notation `x; n` means repeating `x`, `n` times.

References:

[1] “Function composition”
Wikipedia
https://en.wikipedia.org/wiki/Function_composition

[2] “Simply typed lambda calculus”
Wikipedia
https://en.wikipedia.org/wiki/Simply_typed_lambda_calculus

[3] “Turing completeness”
Wikipedia
https://en.wikipedia.org/wiki/Turing_completeness

[4] “Tuple”
Wikipedia
https://en.wikipedia.org/wiki/Tuple

[5] “Calculating Functional Programs”
Jeremy Gibbons
https://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/acmmpc-calcfp.pdf

[6] “Category Theory”
Wikipedia
https://en.wikipedia.org/wiki/Category_theory

[7] “Currying”
Wikipedia
https://en.wikipedia.org/wiki/Currying

[8] “Natural number”
Wikipedia
https://en.wikipedia.org/wiki/Natural_number

[9] “Variadic functions”
Wikipedia
https://en.wikipedia.org/wiki/Variadic_function

[10] “Normal Paths”
Sven Nilsen, 2019
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/normal-paths.pdf

[11] “Imaginary Inverse”
Sven Nilsen, 2020
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/imaginary-inverse.pdf

[12] “Halting problem”
Wikipedia
https://en.wikipedia.org/wiki/Halting_problem

[13] “Function Currying Notation”
Sven Nilsen, 2018
https://github.com/advancedresearch/path_semantics/blob/master/papers-wip/function-currying-notation.pdf